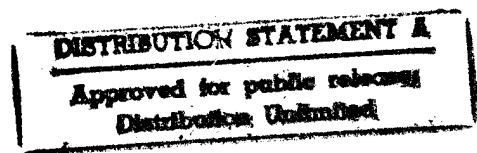
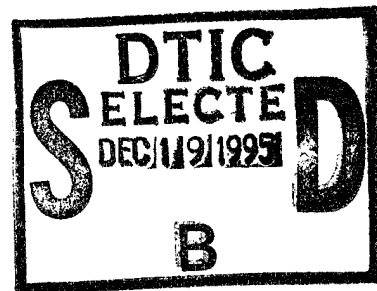




Information Technology -- Programming Language -- The SQL Ada Module
Description Language (SAMEDL)

ISO/IEC 12227

October 1995



19951218 092

DTIC QUALITY INSPECTED 1

Special Report
CMU/SEI-95-SR-018

October 1995

Information Technology -- Programming Language -- The SQL Ada Module
Description Language (SAMEDL)



ISO/IEC 12227

Approved for public release.
Distribution unlimited.

Software Engineering Institute
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213

This report was prepared for the

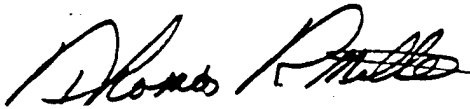
SEI Joint Program Office
HQ ESC/ENS
5 Eglin Street
Hanscom AFB, MA 01731-2116

The ideas and findings in this report should not be construed as an official DoD position. It is published in the interest of scientific and technical information exchange.

Review and Approval

This report has been reviewed and is approved for publication.

FOR THE COMMANDER



Thomas R. Miller, Lt. Col., USAF
SEI Joint Program Office

This work is sponsored by the U.S. Department of Defense.

Copyright © 1995 by Carnegie Mellon University

This work was created in the performance of Federal government Contract Number F19628-95-C-0003 with Carnegie Mellon University for the operation of the Software Engineering Institute, a Federally Funded Research and Development Center. The Government of the United States has a royalty-free government purpose license to use, duplicate, or disclose the work, in whole or part and in any manner, and to have or permit others to do so, for government purposes.

This material may be reproduced by or for the U.S. Government pursuant to the copyright license under the clause at 52.227-7013.

This document is available through Research Access, Inc., 800 Vinial Street, Pittsburgh, PA 15212: Phone: 1-800-685-6510. FAX: (412) 321-2994.

Copies of this document are available through the National Technical Information Service (NTIS). For information on ordering, please contact NTIS directly: National Technical Information Service, U.S. Department of Commerce, Springfield, VA 22161. Phone: (703) 487-4600.

This document is also available through the Defense Technical Information Center (DTIC). DTIC provides access to and transfer of scientific and technical information for DoD personnel, DoD contractors and potential contractors, and other U.S. Government agency personnel and their contractors. To obtain a copy, please contact DTIC directly: Defense Technical Information Center, Attn: DTIC-OCP, 8725 John J. Kingman Road, Suite 0944, Ft. Belvoir, VA 22060-6218. Phone: (703) 767-8019/8021/8022/8023. Fax: 703-767-8032/DSN-427.

Use of any trademarks in this report is not intended in any way to infringe on the rights of the trademark holder.

Contents

Foreword	v
Introduction	vi
1 Scope	1
2 Normative References	1
3 Notations and Structures	2
3.1 Syntax Notation	2
3.2 Semantic Notation	2
3.3 Structure	2
3.4 Examples, Notes and Index Entries	3
4 Design Goals and Language Summary	4
4.1 Design Goals	4
4.2 Language Summary	4
4.2.1 Overview	4
4.2.2 Compilation Units	5
4.2.3 Modules	5
4.2.4 Procedures and Cursors	5
4.2.5 Domain and Base Domain Declarations	6
4.2.6 Other Declarations	6
4.2.7 Value Expressions, Atomic Predicates and Typing	6
4.2.8 Static Typing, Dynamic Schema Definition and Dynamic SQL	7
4.2.9 Standard Post Processing	7
4.2.10 Extensions	7
4.2.11 Default Values in Grammar	7
5 Lexical Elements	8
5.1 Character Set	8
5.2 Lexical Elements, Separators, and Delimiters	8
5.3 Identifiers	9
5.4 Literals and Data Classes	10
5.5 Comments	11
5.6 Reserved Words	11
6 Common Elements	11
6.1 Compilation Units	11
6.2 Context Clause	11
6.3 References	12
6.4 Domain Compatibility	17
6.5 Standard Post Processing	17
6.6 Extensions	18

For	
ed	<input checked="" type="checkbox"/>
ed	<input type="checkbox"/>
tion	<input type="checkbox"/>
on/	
ity Codes	
and/or	
total	

A-1

7 Data Description Language and Data Semantics	19
7.1 Definitional Modules	19
7.1.1 Base Domain Declarations	20
7.1.1.1 Base Domain Parameters	20
7.1.1.2 Base Domain Patterns	21
7.1.1.3 Base Domain Options	22
7.1.2 The SAME Standard Base Domains	23
7.1.3 Domain and Subdomain Declarations	24
7.1.4 Constant Declarations	27
7.1.5 Record Declarations	29
7.1.6 Enumeration Declarations	30
7.1.7 Exception Declarations	31
7.1.8 Status Map Declarations	32
7.2 Schema Modules	33
7.3 Data Conversions	36
8 Abstract Module Description Language	37
8.1 Abstract Modules	37
8.2 Procedures	38
8.3 Non cursor data statements	42
8.4 Cursor Declarations	44
8.5 Cursor Procedures	48
8.6 Input Parameter Lists	51
8.7 Select Parameter Lists	53
8.8 Insert Column Lists and Insert Value Lists	55
8.9 Into_Clause and Insert_From_Clause	57
8.10 Value Expressions	59
8.11 Search Conditions	62
8.12 Status Clauses	62
9 Conformance	63
9.1 Introduction	63
9.2 Claims of Conformance	63
9.2.1 Introduction	63
9.2.2 Conformance via mapping.	63
9.2.2.1 Conformance via SQL module.	63
9.2.2.2 Conformance via embedded SQL Syntax.	64
9.2.3 Conformance via effects.	64
9.2.4 Multiple claims of conformance.	64
9.3 Extensions	64
Annex A SAMeDL_Standard	65
Annex B SAMeDL_System	76
Annex C Standard Support Operations and Specifications	77
C.1 Standard Base Domain Operations	77
C.1.1 All Domains	77
C.1.2 Numeric Domains	78

C.1.3	Int and Smallint Domains	78
C.1.4	Character Domains	79
C.1.5	Enumeration Domains	79
C.1.6	Boolean Functions	80
C.1.7	Operations Available to the Application	80
C.2	Standard Support Package Specifications	82
C.2.1	Interfaces.SQL	82
C.2.2	SQL_Boolean_Pkg	82
C.2.3	SQL_Int_Pkg	83
C.2.4	SQL_Smallint_Pkg	85
C.2.5	SQL_Real_Pkg	87
C.2.6	SQL_Double_Precision_Pkg	89
C.2.7	SQL_Char_Pkg	91
C.2.8	SQL_VarChar_Pkg	94
C.2.9	SQL_Bit_Pkg	97
C.2.10	SQL_Enumeration_Pkg	99
Annex D	Transform Chart	101
Annex E	Glossary	104
Index		107

Foreword

ISO (the International Organization for Standardization) is a worldwide federation of national standards bodies (ISO member bodies). The work of preparing International Standards is normally carried out through ISO technical committees. Each member body interested in a subject for which a technical committee has been established has the right to be represented on that committee. International organizations, governmental and non-governmental, in liaison with ISO, also take part in the work. ISO collaborates closely with the International Electrotechnical Commission (IEC) on all matters of electrotechnical standardization.

Draft International Standards adopted by the technical committees are circulated to the member bodies for approval before their acceptance as International Standards by the ISO Council. They are approved in accordance with ISO procedures requiring at least 75% approval by the member bodies voting.

International Standard ISO/IEC 12227:1994, *Database Programming Language — The SQL Ada Module Description Language SAMeDL*, was prepared by Joint Technical Committee ISO/IEC JTC1, *Information Processing Systems*.

This International Standard contains six annexes of which three are normative and three informative:

- Annex A (normative) : SAMeDL_Standard
- Annex B (normative) : SAMeDL_System
- Annex C (normative) : Standard Base Domain Operations and Specifications
- Annex D (informative) : Transform Chart
- Annex E (informative) : Glossary
- Annex F (informative) : Bibliography

Introduction

The organization of this International Standard is as follows:

1. Clause 1, "Scope," specifies the scope of this International Standard.
2. Clause 2, "Normative References," identifies additional International Standards that, through reference in this International Standard constitute provisions of this International Standard.
3. Clause 3, "Notations and Structures," describes the syntax notation used in and the structure of the clauses of this International Standard.
4. Clause 4, "Design Goals and Language Summary," presents an overview of the SQL Ada Module Description Language.
5. Clause 5, "Lexical Elements," defines the lexical elements of the language.
6. Clause 6, "Common Elements," defines concepts and syntactic categories which are used in the definition of further concepts and syntactic categories of the language.
7. Clause 7, "Data Description Language and Data Semantics," defines that portion of the SQL Ada Module Description Language in which database description and other declarative items are encoded.
8. Clause 8, "Abstract Module Description Language," defines that portion of the SQL Ada Module Description Language in which database manipulation procedures are encoded.
9. Clause 9, "Conformance," defines conformance criteria.
10. Annex A, "SAMeDL_Standard," is a normative Annex containing the text of the predefined SAMeDL Definition Module SAMeDL_Standard.
11. Annex B, "SAMeDL_System," is a normative Annex containing the text of the predefined SAMeDL Definition Module SAMeDL_System. Portions of this module are implementation defined.
12. Annex C, "Standard Support Package Specifications," is a normative Annex containing Ada text declaring the standard support packages.
13. Annex D, "Transform Chart," is an informative annex giving in a tabular form a description of string-to-string transformation functions used in the definition of the SAMeDL.
14. Annex E, "Glossary," is an informative annex giving explanations of terms used in this International Standard.
15. Annex F is a Bibliography.

This International Standard is based on and defined with respect to the International Standards for *Database Language SQL*, ISO 9075:1992 and *Programming Language Ada*, ISO 8652:1987 (endorsement by ISO of ANSI/MIL-STD-1815A-1983 and AFNOR NF Z 65-700).

SQL/Ada Module Description Language SAMeDL

1 Scope

This International Standard specifies the syntax and semantics of a database programming language, the SQL Ada Module Description Language, SAMeDL. Texts written in the SAMeDL describe database interactions which are to be performed by database management systems (DBMS) implementing Database Language SQL. The interactions so described and so implemented are to be performed on behalf of application programs written in Programming Language Ada.

The SAMeDL is not a Programming Language; it may be used solely to specify application program database interactions and solely when those interactions are to occur between an Ada application program and an SQL DBMS.

The SAMeDL is defined with respect to Entry Level SQL. Therefore, all inclusions by reference of text from ISO/IEC 9075:1992 include all applicable Leveling Rules for Entry Level SQL.

This International Standard does not define the Programming Language Ada nor the Database Language SQL. Therefore, ISO/IEC 8652:1995 takes precedence in all matters dealing with the syntax and semantics of any Ada construct contained, referred or described within this International Standard; similarly, ISO/IEC 9075:1992 takes precedence in all matters dealing with the syntax and semantics of any SQL construct contained, referred or described within this International Standard.

Note: The SAMeDL is an example of an Abstract Modular Interface. A reference model for programming language interfaces to database management systems, which includes a description of Abstract Modular interfaces, can be found in reference [2].

2 Normative References

The following International Standards contain provisions which, through reference in this text, constitute provisions of this International Standard. At the time of publication, the editions indicated were valid. All International Standards are subject to revision, and parties to agreements based on this International Standard are encouraged to investigate the possibility of applying the most recent editions of the International Standards indicated below. Members of IEC and ISO maintain registers of currently valid International Standards.

- ISO/IEC 9075:1992, *Information technology -- Database languages -- SQL*.
- ISO/IEC 8652:1995, *Information technology -- Programming languages -- Ada*

3 Notations and Structures

3.1 Syntax Notation

The context-free syntax of the language is described using a simple variant of the variant of BNF used in the description of Ada, (1.5 in ISO/IEC 8652:1995). The variation is:

- Underscores are preserved when using the name of a syntactic category outside of a syntax rule (1.5(6) of ISO/IEC 8652:1995).
- The italicized prefixes *Ada* and *SQL*, when appearing in the names of syntactic categories, indicate that an Ada or SQL syntactic category has been incorporated into this document. For example, the category *Ada_identifier* is identical to the category identifier as described in 2.3 of ISO/IEC 8652:1995; whereas the category *SQL_identifier* is identical to the category identifier as described in 5.4 of ISO/IEC 9075:1992.
- Numerical suffixes attached to the names of syntactic categories are used to distinguish appearances of the category within a rule or set of rules. An example of this usage is given below.

3.2 Semantic Notation

The meaning of a SAMeDL compilation unit (except where specified as implementation-defined) is given by:

- An Ada compilation unit, conforming to ISO/IEC 8652:1995.
- A module conforming to the SQL standard as given by clause 12 of ISO/IEC 9075:1992.
- Interface rules, concerning the relationship between the SQL and Ada texts.

The semantics of SAMeDL constructs are given in part by collections of string transformers that produce Ada and SQL texts from SAMeDL input.

Note: A quick reference to these transformers appears in Annex D.

The effects of these transformers are described through the use of sample input strings. Those strings are written in a variant of the syntax notation. For example, the syntax of an *input_parameter* (see 8.6) is given by:

```
identifier_1 [named_phrase] : domain_reference [not null]
```

A representative input parameter declaration is given by

```
Id_1 [named Id_2] : Id_3 [not null]
```

It is then possible to discuss the four variants of input parameters (the variants described by the presence or absence of optional phrases) in a single piece of text.

3.3 Structure

The remainder of this International Standard is structured in the following way. Clause 4 contains a descriptive overview of the goals and concepts of the SAMeDL. Clause 5 defines the lexical structure of the SAM-

eDL including the rules for identifier and literal formation and the list of reserved words. Clauses 6, 7 and 8 define the syntax and semantics of the SAMeDL. Each subclause of those clauses adheres to the following general format.

- The purpose of the item being defined by the subclause is introduced.
- The syntax of the item being defined is given in the notation described earlier.
- Abstract (non-context free) syntactical rules governing formation of instances of the item being defined are given, if applicable.
- The semantics of the item being defined are given. These semantics are given under the headings **Ada Semantics**, **SQL Semantics** and **Interface Semantics**, as appropriate.

3.4 Examples, Notes and Index Entries

Many of the subclauses of this International Standard are illustrated with examples. These examples are introduced by the words: "*Note: Examples*" on a line by themselves and are terminated by the words "End Examples," likewise appearing on a line by themselves.

This is an example of an example.

Other notes also appear in this Standard, introduced by the word *Note*. Both kinds of note are informative only and do not form part of the standard definition of the SAMeDL.

Items in the grammar that contain underscores are represented in the Index by a corresponding entry without underscores. For example, the "Ada identifier" entry in the index contains the page numbers of occurrences of both "Ada_identifier" and "Ada identifier".

4 Design Goals and Language Summary

4.1 Design Goals

The SQL Ada Module Description Language (SAMEDL) is a Database Programming Language designed to automate the construction of software conformant to the SQL Ada Module Extensions (SAME) application architecture. This architecture is described in the document, *Guidelines for the Use of the SAME* [1].

The SAME is a *modular* architecture. It uses the concept of a Module as defined in 4.16 and 12 of ISO/IEC 9075:1992. As a consequence, a SAME-conformant Ada application does not contain embedded SQL statements and is not an embedded SQL Ada program as defined in 19.3 of ISO/IEC 9075:1992. Such a SAME-conformant application treats SQL in the manner in which Ada treats all other languages: it imports complete functional modules, not language fragments.

Modular architectures treat the interaction of the application program and the database as a design object. This results in a further isolation of the application program from details of the database design and implementation and improves the potential for increased specialization of software development staff.

Ada and SQL are vastly different languages: Ada is a Programming Language designed to express algorithms, while SQL is a Database Language designed to describe desired results. Text containing both Ada and SQL is therefore confusing and difficult to maintain. SAMEDL is a Database Programming Language designed to support the goals and exploit the capabilities of Ada with a language whose syntax and semantics is based firmly in SQL. Beyond modularity, the SAMEDL provides the application programmer the following services:

- An abstract treatment of null values. Using Ada typing facilities, a safe treatment of missing information based on SQL is introduced into Ada database programming. The treatment is safe in that it prevents an application from mistaking missing information (null values) for present information (non-null values).
- Robust status code processing. SAMEDL's Standard Post Processing provides a structured mechanism for the processing of SQL status parameters.
- Strong typing. SAMEDL's typing rules are based on the strong typing of Ada, not the permissive typing of SQL.
- Extensibility. The SAMEDL supports a class of user extensions. Further, it controls, but does not restrict, implementation defined extensions.

4.2 Language Summary

4.2.1 Overview

The SAMEDL is designed to facilitate the construction of Ada database applications that conform to the SAME architecture as described in [1]. The SAME method involves the use of an abstract interface, an abstract module, a concrete interface, and a concrete module. The abstract interface is a set of Ada package specifications containing the type and procedure declarations to be used by the Ada application program. The abstract module is a set of bodies for the abstract interface. These bodies are responsible for invoking

the routines of the concrete interface, and converting between the Ada and the SQL data and error representations. The concrete interface is a set of Ada specifications that define the SQL procedures needed by the abstract module. The concrete module is a set of SQL procedures that implement the concrete interface.

Within this International Standard, the concrete module of [1] is called an SQL module and its contents are given under the headings **SQL Semantics** within the clauses of this specification. The abstract modules of [1] are given under the heading **Ada Semantics** within the clauses of this specification.

4.2.2 Compilation Units

A *compilation unit* consists of one or more modules. A module may be either a definitional module containing shared definitions, a schema module containing table, view, and privilege definitions, or an abstract module containing local definitions and procedure and cursor declarations.

4.2.3 Modules

A *definitional module* contains the definitions of base domains, domains, constants, records, enumerations, exceptions, and status maps. Definitions in definitional modules may be seen by other modules.

A *schema module* contains definitions of tables, views, and privileges and other so-called schema or data definition language (DDL) constructs. These may appear within procedure declarations, in which case they become “executable DDL” and may be invoked from an Ada application.

An *abstract module* defines (a portion of) an application’s interface to the database: it defines SQL services needed by an Ada application program. An abstract module may contain procedure declarations, cursor declarations, and definitions such as those that may appear in a definitional module. Definitions in an abstract module, however, may not be seen by other modules. Any SQL statement, including dynamic SQL statements and schema manipulation statements, but *excluding schema definition statements*, which are restricted to schema modules, may appear in abstract modules.

4.2.4 Procedures and Cursors

A *procedure* declaration defines a basic database operation. The declaration defines an Ada procedure declaration and a corresponding SQL procedure. A SAMeDL procedure consists of a single statement along with an optional input parameter list and an optional status clause. The input parameter list provides the mechanism for passing information to the database at runtime. A statement in a SAMeDL procedure may be any statement defined by SQL (see 12.5 of ISO/IEC 9075:1992) or an implementation-defined extended statement. Generally speaking, SAMeDL statements differ from SQL statements in that strongly typed value expressions and atomic predicates are used. See 4.2.7, below.

In contrast to the language of ISO/IEC 9075:1992, the procedures that operate on cursors, procedures containing either an open, fetch, close, update positioned or delete positioned statement, are packaged with the declaration of the cursor upon which they operate, thereby improving readability. Further, if no procedure containing an open, fetch or close statement is explicitly given in a cursor declaration, the language provides such procedures implicitly, thereby improving writeability (ease of use).

4.2.5 Domain and Base Domain Declarations

Objects in the language have an associated *domain*, which characterizes the set of values and applicable operations for that object. In this sense, a domain is similar to an Ada type. A SAMeDL domain is *not* an SQL domain.

A *base domain* is a template for defining domains. A base domain declaration consists of a set of parameters, a set of patterns and a set of options. The parameters are used to supply information needed to declare a domain or subdomain derived from the base domain. Patterns contain templates for the generation of Ada code to support the domain in Ada applications. This code generally contains type declarations and package instantiations. Options contain information needed by the compiler. Parameters may be used in the patterns and options and their values may be referenced in other statements.

Base domains are classified according to their associated data class. A data class is either integer, fixed, float, enumeration, or character. A numeric base domain has a data class of either integer, fixed, or float. An enumeration base domain has a data class of enumeration, and defines both an ordered set of distinct enumeration literals and a bijection between the enumeration literals and their associated database values. A character base domain has a data class of character.

SQL provides for the storage of date and time information (“temporal data”) in its databases. It does not allow this data to pass across the application program interface; the application must deal with temporal data as character strings and cast between the string and internal representations in the SQL statements. The SAMeDL supports temporal data as statement and as database data and the base domain facility recognizes that certain data can not appear at the interface.

4.2.6 Other Declarations

Certain SAMeDL declarations are provided as a convenience for the user. For example, *constant* declarations name and associate a domain with a static expression. *Record* declarations allow distinct procedures to share types. An *exception* declaration defines an Ada exception declaration with the same name.

4.2.7 Value Expressions, Atomic Predicates and Typing

Value expressions are formed and evaluated according to the rules of SQL, with the exception that the strong typing rules are based on those of Ada. In the typing rules of the SAMeDL, the domain acts as an Ada type in a system without user defined operations. Strong typing necessitates the introduction of domain conversions. The SQL `CAST` operator is used. For compatibility with an earlier version of SAMeDL, a syntax resembling Ada type conversion is also provided. If that Ada-like syntax is *not* used, the concrete syntax of value expressions as defined in this specification is identical to that of SQL except for the removal of the colon in parameter references. The abstract syntax is different, primarily in that it applies strong typing. Similarly, the concrete syntax of predicates is that of SQL but these predicates are strongly typed.

4.2.8 Static Typing, Dynamic Schema Definition and Dynamic SQL.

Following the principles of Ada, the SAMeDL is statically typed. SQL, in contrast, allows for run time modification of its set of data definitions, the schema. Schema modules may contain declarations of tables or views that anticipate the state of the SQL schema at run time, rather than the state of the SQL schema at the time at which the schema module is written or compiled.

Dynamic SQL statements take parameters whose types may not be known until run time. It is the programmer's responsibility to provide parameters of the correct types. The SAMeDL presents this paradigm directly to the Ada programmer.

4.2.9 Standard Post Processing

Standard post processing is performed after the execution of an SQL procedure but before control is returned to the calling application procedure. The *status clause* from a SAMeDL procedure declaration attaches a *status mapping* to the application procedure. That status mapping is used to process SQL status data in a uniform way for all procedures and to present SQL status codes to the application in an application-defined manner, either as a value of an enumerated type, or as a user defined exception. SQL status codes not specified by the status map result in a call to a standard database error processing procedure and the raising of the predefined SAMeDL exception, `SQL_Database_Error`. This prevents a database error from being ignored by the application.

4.2.10 Extensions

The data semantics of the SAMeDL may be extended without modification to the language by the addition of user-defined base domains. For example, a user-defined base domain of `DATE` may be included without modification to the SAMeDL.

DBMS specific (i.e., non-standard) operations may also be included into the SAMeDL. Such additions to the SAMeDL are referred to as extensions. Schema elements, table elements, statements, query expressions, query specifications, and cursor statements may be extended. The modules, tables, views, cursors, and procedures that contain these extensions are marked (with the keyword **extended**) to indicate that they go outside the standard.

4.2.11 Default Values in Grammar

Obvious but over-ridable defaults are provided in the grammar. For example, open, close, and fetch statements are essential for a cursor, but their form may be deduced from the cursor declaration. The SAMeDL will therefore supply the needed open, close, and fetch procedure declarations if they are not supplied by the user.

5 Lexical Elements

The text of a compilation is a sequence of lexical elements, each composed of characters from the basic character set. The rules of composition are given in this chapter.

5.1 Character Set

The only characters allowed in the text of a compilation are the basic characters and the characters that make up character literals (described in 5.4 of this specification). Each character in the basic character set is represented by a graphical symbol.

```
basic_character ::=
    upper_case_letter | lower_case_letter | digit |
    special_character | space_character
```

The characters included in each of the above categories of the basic characters are defined as follows:

1. upper_case_letter
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
2. lower_case_letter
a b c d e f g h i j k l m n o p q r s t u v w x y z
3. digit
0 1 2 3 4 5 6 7 8 9
4. special_character
' () * + , - . / : ; < = > _ |
5. space_character

5.2 Lexical Elements, Separators, and Delimiters

The text of each compilation is a sequence of separate lexical elements. Each lexical element is either an identifier (which may be a reserved word), a literal, a comment, or a delimiter. The effect of a compilation depends only on the particular sequences of lexical elements excluding the comments, if any, as described in this chapter. Identifiers, literals, and comments are discussed in the following clauses. The remainder of this clause discusses delimiters and separators.

An explicit separator is required to separate adjacent lexical elements when, without separation, interpretation as a single lexical element is possible. A separator is any of a space character, a format effector, or the end of a line. A space character is a separator except within a comment or a character literal. Format effectors other than horizontal tabulation are always separators. Horizontal tabulation is a separator except within a comment.

The end of a line is always a separator. The language does not define what causes the end of a line.

One or more separators are allowed between any two adjacent lexical elements, before the first lexical element of each compilation, or after the last lexical element of each compilation. At least one separator is required between an identifier or a numeric literal and an adjacent identifier or numeric literal.

VERSION 2

A delimiter is one of the following special characters

`() * + , - . / : ; < = > |`

or one of the following compound delimiters each composed of two adjacent special characters

`=> .. := <> >= <=`

Each of the special characters listed for single character delimiters is a single delimiter except if that character is used as a character of a compound delimiter, a comment, or a literal.

Each lexical element shall fit on one line, since the end of a line is a separator. The single quote and underscore characters, as well as two adjacent hyphens, are not delimiters, but may form part of other lexical elements.

5.3 Identifiers

`identifier ::= SQL_actual_identifier`

The length restrictions that apply to *SQL_identifiers* (see 5.2, syntax rules 8, 9 and 5.4 syntax rule 3 of ISO/IEC 9075:1992) *do not apply* to SAMeDL identifiers. Whenever two identifiers are deemed equivalent, it is in the sense of the rules of SQL (see 5.2, syntax rules 10 through 14 of ISO/IEC 9075:1992).

Note. An *SQL_actual_identifier* is either a *delimited_identifier* or a *regular_identifier*. The form of an *SQL_regular_identifier* is essentially the same as the Ada identifier, except that a *regular_identifier* may end in an underscore. A *delimited_identifier* is *any* character string within a pair of double quote characters ("). *Delimited identifiers* provide a means of using tokens that would otherwise be reserved words (see 5.6 of this specification) as identifiers. Thus *fetch* is a reserved word, but the construct

```
procedure "fetch" is fetch;
```

defines a procedure named "fetch" that contains a *fetch* statement.

Identifier equivalence in SQL is similar to Ada identifier equivalence for regular identifiers. Equivalence for delimited identifiers is case sensitive. Equivalence of regular identifiers with delimited identifiers proceeds by considering the regular identifier to be all upper case and then doing a case sensitive comparison to the delimited identifier. So a column named *Status* is not identified by the delimited identifier "Status" but is identified by the delimited identifier "STATUS". *end Note*

Ada Semantics

Let *ident* be an identifier. Define *AdaID(ident)* by

```
AdaID(ident) =  ident  if ident is a regular identifier
                id     if ident is the delimited identifier "id"
```

Note: If *ident* is an identifier, *AdaID(ident)* is not necessarily an *Ada_identifier*.

SQL Semantics

Let *SQL_NAME* be a function on identifiers into the set of *SQL_identifiers* with the property that *SQL_NAME(Id_1)* and *SQL_NAME(Id_2)* shall be equivalent if and only if *Id_1* and *Id_2* are equivalent (see above).

5.4 Literals and Data Classes

Literals follow the SQL literal syntax (5.3 of ISO/IEC 9075:1992). There are eight classes into which literals are placed based on their lexical properties: **character**, **bit**, **integer**, **fixed**, **float**, **date**, **time**, **timestamp**, **interval** and **enumeration**.

```

literal ::=
    database_literal | enumeration_literal

database_literal ::=
    character_literal
    | SQL_bit_string_literal
    | SQL_hex_string_literal
    | [+ | -] numeric_literal
    | SQL_date_literal
    | SQL_time_literal
    | SQL_timestamp_literal
    | SQL_interval_literal

numeric_literal ::=
    integer_literal | fixed_literal | float_literal

character_literal :: ' {character} '

character ::= implementation defined

integer_literal ::= digit {digit}

fixed_literal ::=
    integer_literal . integer_literal
    | . integer_literal
    | integer_literal .

float_literal ::= fixed_literal exp [+ | -] integer_literal
exp ::= e | E

enumeration_literal ::= identifier

```

1. Each literal has an associated data class, denoted **DATACLASS(L)**. In particular:

if <i>L</i> is a character_literal then DATACLASS(L) is character	
SQL_bit_string_literal	bit
SQL_hex_string_literal	bit
integer_literal	integer
fixed_literal	fixed
float_literal	float
SQL_date_literal	date
SQL_time_literal	time
SQL_timestamp_literal	timestamp
SQL_interval_literal	interval
enumeration_literal	enumeration

Literals of a temporal data class are called temporal literals. (see data_class in 7.1.1.1 of this specification).

2. If *Id* is an enumeration literal, then **AdaID(id)** shall be an *Ada_identifier*. In particular, an enumeration literal shall not be an *Ada character literal*.
3. The qualifier of an *SQL_interval_literal*, *IL*, denoted **QUALIFIER(IL)**, is the value of its *SQL_interval_qualifier*. See 5.3 and 10.1 of ISO/IEC 9075:1992.
4. The single quote or "tic" character can be included in a character_literal by duplication in the usual way. Thus the string of tics: `''''` represents a character_literal of length one containing the single quote as its only character.

5.5 Comments

A comment starts with two adjacent hyphens and extends up to the end of the line. A comment can appear on any line of a module. The presence or absence of comments has no influence on whether a module is legal or illegal. Furthermore, comments do not influence the meaning of a module; their sole purpose is the enlightenment of the human reader.

5.6 Reserved Words

The following is the list of the SAMeDL reserved words:

ABSTRACT	ALL	AND	ANY	AS	ASC
AUTHORIZATION	AVG	BASE	BETWEEN	BODY	BY
CHECK	CLASS	CLOSE	COMMIT	CONSTANT	CONVERSION
COUNT	CURRENT	CURSOR	DATA	DBMS	DECLARE
DEFAULT	DEFINITION	DELETE	DERIVED	DESC	DISTINCT
DOMAIN	END	ENUMERATION	ESCAPE	EXISTS	EXCEPTION
EXTENDED	FETCH	FOR	FOREIGN	FROM	GRANT
GROUP	HAVING	IMAGE	IN	INSERT	INTO
IS	KEY	LIKE	MAP	MAX	MIN
MODULE	NAME	NAMED	NEW	NOT	NULL
OF	ON	OPEN	OPTION	OR	ORDER
OUT	PATTERN	POS	PRIMARY	PRIVILEGES	PROCEDURE
PUBLIC	RAISE	RECORD	REFERENCES	ROLLBACK	SCALE
SCHEMA	SELECT	SET	SOME	STATUS	SUBDOMAIN
SUM	TABLE	TO	TYPE	UNION	UNIQUE
UPDATE	USE	USER	USES	VALUES	VIEW
WHERE	WITH	WORK			

6 Common Elements

6.1 Compilation Units

A *compilation unit* is the smallest syntactic object that can be successfully compiled. It consists of a sequence of one or more modules.

```

compilation_unit ::=
  module {module}

module ::=
  definitional_module | abstract_module | schema_module

```

6.2 Context Clause

The context clause is a means by which a module gains visibility to names defined in other modules. The syntax and semantics of context clauses are similar to the syntax and semantics of Ada context clauses (8.4, 10.1.1 of ISO/IEC 8652:1995) but there are differences.

```

context ::=
  context_clause {context_clause}

context_clause ::=
  with_clause | use_clause | with_schema_clause

with_clause ::=
  with module_name [as_phrase]
    {, module_name [as_phrase]} ;

use_clause ::=
  use module_name {, module_name} ;

with_schema_clause ::=
  with schema schema_module_name [as_phrase]
    {, schema_module_name [as_phrase] } ;

module_name ::=
  identifier

schema_module_name ::=
  identifier

as_phrase ::=
  as identifier

```

1. Consider the following with_clause and with_schema_clause:

```

      with M [as N1];
  with schema S [as N2];

```

In these clauses, M shall be the name of a definitional module and S the name of a schema module. The name M of the definitional module is said to be *exposed* if the as_phrase is not present in the context_clause; otherwise the name M is *hidden* and the name N₁ is the exposed name of M. Similar comments apply to S and N₂. The name of a module (see 7.1, 7.2, and 8.1 of this specification) is its exposed name within the text of that module. Within the text of any module, no two exposed module names shall be the same.

2. A module_name in a use_clause shall be the exposed name of a definitional_module that is an operand of a prior with_clause.
3. The scope of a with_clause or use_clause in the context of a module (see 7.1, 7.2, and 8.1 of this specification) is the text of that module.
4. Only an abstract or schema module context may contain a with_schema_clause.

Note: As a consequence of these definitions, abstract modules cannot be brought into the context of (withed by) another module.

6.3 References

The rules concerning the meaning of references are modeled on those of Ada and those of SQL. As neither module nesting nor program name overloading occurs, these rules are fairly simple, and are therefore listed. For the purposes of this clause, an *item* is either:

- A definitional module (See 7.1), an abstract module (8.1), or a schema module (7.2 of this specification).
- A procedure (See 8.2), a cursor (8.4), or a procedure within a cursor (8.5 of this specification).
- Anything in the syntactic category 'definition' as given by 7.1 of this specification. This includes base domains, domains, subdomains, enumerations, constants, records, and status maps.
- A domain parameter (See 7.1.1.1 of this specification.)

VERSION 2

- An enumeration literal within an enumeration (See 5.4 and 7.1.6 of this specification).
- An exception (7.1.7 of this specification).
- An input_parameter of a procedure or cursor declaration (See 8.2, 8.4, and 8.6 of this specification).
- A table or view defined within a schema module or a temporary table declared within an abstract module. (see 7.2 and 8.1 of this specification).
- A column defined within a table (see 7.2 of this specification).

A location within the text of a module is said to be a *defining* location if it is the place of

- The name of an item within the item's declaration. (*Note:* This includes enumeration literals within the declaration of an enumeration and domain parameters within the declaration of a domain.)
- The name of a table or view in an *SQL_from_clause*. For the definitions of *exposed* table name and the *scope* of a table name see 6.3 in ISO 9075:1992.
- The name of the target table of an insert, update, or delete statement.
- A schema_module_name or module_name in a context_clause.

Text locations not within comments that are not defining locations are *reference* locations. An identifier that appears at a reference location is a reference to an item. The meaning of that reference in that location, that is, the identity of the item referenced, is defined by the rules of this clause. When these rules determine more than one meaning for an identifier, then all items referenced shall be enumeration literals.

```
module_reference ::= identifier
schema_module_reference ::= schema_module_name | identifier
base_domain_reference ::= [module_reference.]identifier
domain_reference ::= [module_reference.]identifier
domain_parameter_reference ::= domain_reference.identifier
subdomain_reference ::= [module_reference.]identifier
enumeration_reference ::= [module_reference.]identifier
enumeration_literal_reference ::= [module_reference.]identifier
exception_reference ::= [module_reference.]identifier
constant_reference ::= [module_reference.]identifier
record_reference ::= [module_reference.]identifier
procedure_reference ::= [module_reference.]identifier
cursor_reference ::= [module_reference.]identifier
cursor_proc_reference ::= [cursor_reference.]identifier
input_reference ::= [procedure_reference.]identifier
                  | [cursor_proc_reference.]identifier
status_reference ::= [module_reference.]identifier
table_name ::= [schema_module_reference.]SQL_identifier
             | MODULE.SQL_identifier
column_name ::= SQL_identifier
```

VERSION 2

```
column_reference ::= [qualifier.]column_name.  
qualifier ::= table_name  
            | SQL_correlation_name
```

Note: A table_name differs from an SQL_table_name only in that a schema_module_reference replaces a schema name.

A reference is a simple name (an identifier) optionally preceded by a prefix: a sequence of as many as three identifiers, separated by dots.

For the purposes of this clause, the “text of a cursor” does not include the text of the procedures, if any, contained in the cursor. A dereferencing rule is said to “determine a denotation” for a reference if it either (i) specifies an item to which the reference refers, or (ii) determines that the reference is not valid.

Note: Unlike the Ada dereferencing rules (see 8.2 and 8.3 in ISO/IEC 8652:1995), the SAMeDL rules treat the prefix as a whole, not component by component.

Prefix Denotations

The prefix of a reference shall denote one of the following:

- An abstract module, procedure, cursor or cursor procedure, but only from within the text of the abstract module, procedure, cursor, or cursor procedure.
- A table or view, if the table or view is in scope at the location in which the reference appears.
- A domain.
- A definitional or schema module.

Note: As a consequence of the rule given earlier, that all meanings of an identifier with multiple meanings must be enumeration literals, a prefix may have at most one denotation or meaning, as it may not denote an enumeration literal.

Let L be the reference location of prefix P . Let X , Y , and Z be simple names. Then

1. If L is within the text of a cursor procedure U , then P denotes
 - a. The cursor procedure U if either
 1. P is of the form X and X is the simple name of U ; or
 2. P is of the form $X.Y$; X is the name of the cursor containing L (and therefore also U); in which case Y shall be the simple name of U else the prefix is not valid; or
 3. P is of the form $X.Y.Z$; X is the name of the module containing L ; Y is the name of the cursor containing L (and therefore also U); in which case Z shall be the simple name of U else the prefix is not valid;
 - b. The table T being updated in a cursor_update_statement, if the statement within the cursor procedure containing L is a cursor_update_statement and either
 1. P is of the form X and X is the simple name of T ; or
 2. P is of the form $X.Y$; X is an exposed name for the schema module S containing the declaration of T and Y is the simple name of T .
2. If rule 1 does not determine a denotation for P , then P denotes
 - a. The cursor or procedure R , if L is within the text of R and either
 1. P is of the form X and X is the simple name of R ; or

VERSION 2

2. P is of the form $X.Y$; X is the name of the module containing L (and therefore also R); Y is the simple name of R ;
- b. The table T (or view V), if P denotes the table T (or view V) according to 6.3 of ISO 9075:1992, *unless* P is of the form $X.Y$ and X is not the keyword **MODULE**, in which case X is the exposed name of a schema module containing the table T and Y is the simple name of T .
3. If rules 1 and 2 do not determine a denotation for P , then P denotes the domain R ,
 - a. If P is of the form X and X is the simple name of R and the declaration of R appears in the module containing L and precedes L within that module; or
 - b. P is of the form $X.Y$; X is the exposed name of the module containing the declaration of R and Y is the simple name of R .
4. If none of the above rules determines a denotation for P , then P is a simple name that denotes the
 - a. Definitional module M if either
 1. L is in the scope of a `with_clause` exposing P as the name of M ; or
 2. L is in the definitional module M and P is the name of M .
 - b. Schema module S if either
 1. L is in the scope of a `with_schema_clause` exposing P as the name of S ; or
 2. L is in an abstract module whose authorization clause identifies S and P is the name of S ;
 - c. Abstract module M if L is within the text of M and P is the name of M ;
 - d. Domain D , if D is declared within a module N such that there is a use clause for N in the module containing L , and P is the name of D .

Denotations of Full Names

Let L be the location of a reference ld . Then ld is a reference to the item lm if lm is not a module, procedure, cursor or cursor procedure, or table and

1. ld is of the form $P.X$ where X is the name of lm and P is a prefix denoting
 - a. A definitional module containing the declaration of lm ;
 - b. The abstract module, M , in which L appears, and lm is declared in M at a text location that precedes L ;
 - c. The procedure, cursor, or cursor procedure that contains L , and lm is an input parameter to that procedure, cursor, or cursor procedure;
 - d. A table, in which case lm is a column within that table;
 - e. A schema module, in which case lm is a table within that module.
 - f. A domain, in which case lm is a parameter in that domain.
 - g. The keyword **MODULE**, in which case lm is the local table name of a temporary table.
2. ld is of the form X and X is the name of lm . Then
 - a. L appears in a cursor, procedure, or cursor procedure. (See 8.4, 8.2, and 8.5 of this specification) and
 1. lm is an input parameter to that cursor, procedure, or cursor procedure;

2. *Im* is a column of one of the tables in scope of *L*;
- b. If rule (a) does not determine a denotation for *Id*, then *Im* is
- c. declared in the module containing *L* at a location preceding *L*;
- d. If neither rule (a) nor (b) determines a denotation for *Id*, then *Im* is declared within a module *M* such that the module containing *L* has a use clause for *M*.

Note: An item *Im* is *visible* at location *L* if there exists a name *Id* (either simple or preceded by a prefix) such that if *Id* were at location *L*, then *Id* would be a reference to *Im*.

Note: Examples

The following clause contains examples of the disambiguation of prefixes, with the applicable rule in a comment. At the point in Proc1 marked by Note1, the declaration of constant Inp2 is hidden by the input parameter Inp2: that constant would have to be qualified by the prefix "Abmod" to be visible. At the point in Proc2 marked by Note2, if COL is the name of a column of visible table TABNAME, then that reference is ambiguous. A reference to the input parameter would have to be "Proc2.Col", while a reference to the table column would have to be "TABNAME.COL". Finally, Dom1 is visible at Note3 since Defmod is in both a **with** and **use** clause in Abmod. Without the **use** clause, the reference at Note3 would have to be to "Defmod.Dom1".

```

with SAMEDL_Standard;
use SAMEDL_Standard;
definition module Defmod is
  constant Newfirst is 0;
  constant Newlast is 999;
  domain Dom1 is new SQL_Int (First => Newfirst,
                              Last => Defmod.Newlast); -- 4a(ii)
end Defmod;
with Defmod, SAMEDL_Standard;
use Defmod, SAMEDL_Standard;
with schema Sname1;
abstract module Abmod is
  authorization Sname2
  constant Newfirst is 0;
  constant Inp2 : Dom1 is 1; -- Note3
  domain Dom is new SQL_Int (First => Abmod.Newfirst, -- 4c
                              Last => Defmod.Newlast); -- 4a(i)
  declare local temporary table TEMTAB
    (COL : DOM1);
  procedure Proc (Inp1 : Dom; Inp2 : Dom) is
    insert into TABNAME
      select Proc.Inp1, -- 2a(i)
             Abmod.Proc.Inp2 -- 2a(ii)
      from TABNAME
  ;
  cursor Curse
  for
    select COL
    from Sname1.TAB -- 4b(i)
  ;
  is
    procedure Proc1 (Inp1 : Dom; Inp2 : Dom; Inp3 : Dom) is
      update Sname2.TABNAME -- 4b(ii)
      set COL1 = Proc1.Inp1, -- 1a(i)
          COL2 = Curse.Proc1.Inp2, -- 1a(ii)
          COL3 = Abmod.Curse.Proc1.Inp3, -- 1a(iii)
          TABNAME.COL4 = Inp1, -- 2b(i)
          Sname2.TABNAME.COL5 = Inp2 -- 2b(ii) : Note1
    ;
  end Curse;
  procedure Proc2 (Col : Dom) is
    insert into TABNAME
      select COL -- Note2
    from TABNAME
  ;

```

```

procedure Proc3 is
  insert into TABNAME
    select COL      -- Note2
    from MODULE.TEMPTAB  1g
;
cursor Cursel
for
  select COL
  from Sname2.TABNAME  -- 4b(ii)
;
end Abmod;
End Examples

```

6.4 Domain Compatibility

An expression E is compatible with a domain D if

1. $\text{DOMAIN}(E) \neq \text{NO_DOMAIN}$ then $\text{DOMAIN}(E)=D$; otherwise
2. D is an enumeration domain and E evaluates to a literal of D ; otherwise
3. If $\text{DATACLASS}(D)$ is not numeric, then $\text{DATACLASS}(E) = \text{DATACLASS}(D)$ and if $\text{DATACLASS}(D)$ is **interval** then $\text{QUALIFIER}(D) = \text{QUALIFIER}(E)$.
4. If $\text{DATACLASS}(D)$ is numeric, either both of $\text{DATACLASS}(E)$ and $\text{DATACLASS}(D)$ are **integer** or neither is.

Note: The rule concerning intervals reproduces the concept of comparability of temporal data types from ISO/IEC 9075:1992. When used in a context requiring a domain, e.g., assignment to a column, a numeric literal conforms to the rule of Ada.

A pair of expressions E_1 and E_2 are said to be domain compatible

1. $\text{DOMAIN}(E_1) \neq \text{NO_DOMAIN}$, implies E_2 is compatible with $\text{DOMAIN}(E_1)$; otherwise,
2. $\text{DOMAIN}(E_2) \neq \text{NO_DOMAIN}$, implies E_1 is compatible with $\text{DOMAIN}(E_2)$; otherwise,
3. $\text{SQL}_{VE}(E_1)$ and $\text{SQL}_{VE}(E_2)$ are comparable as defined in ISO/IEC 9075:1992.

Note: Any two numeric literals will be domain compatible, in contrast to the situation when one of the expressions has a domain, as described above.

See sections 7.1.3 and 8.10 of this specification.

6.5 Standard Post Processing

Standard post processing is the processing that is done after execution of an SQL procedure, but before control is returned to the calling application. That processing is described as follows:

1. The SQL status parameter in an SQL procedure call is **SQLSTATE** unless a status clause appears in the procedure declaration that references a status map specifying **sqlcode**, in which case the SQL status parameter is **SQLCODE**. See 7.1.8 and 8.12 of this specification.

Note: **SQLSTATE** is the preferred status parameter. **SQLCODE** is a deprecated feature of ISO/IEC 9075:1992.

If a status map is attached to the procedure via a status clause (see 8.12 of this specification), then if the value of the status parameter appears in the left hand side of some **status_assignment** in that status map (possibly via the equivalences in 7.1.8 of this specification) then the

Ada procedure's status parameter is set to the value of the right hand side of that status_assignment, if that right hand side is an enumeration_literal; if that right hand side is a **raise** statement, then the named *Ada_exception* is raised. This is not considered an error condition in the sense of the next paragraph. In particular, SQL_Database_Error_Pkg.Process_Database_Error is not called.

2. If the value of the SQL status parameter does not appear in the left hand side of any status_assignment in the map attached to the procedure *or* there is no status map attached to the procedure and the SQL status parameter indicates a condition other than successful completion, then an error condition exists. In this case the parameterless procedure SQL_Database_Error_Pkg.Process_Database_Error is called. Upon return from that procedure, the exception SAMEDL_Standard.SQL_Database_Error is raised.

Note: Successful completion is indicated by an SQLSTATE value of "00000" or an SQLCODE value of 0. In particular, warning values, SQLSTATE values "01xxx" or positive SQLSTATE values other than 100, are not considered to indicate successful completion in the sense of this paragraph.

6.6 Extensions

Extended tables, views, modules, procedures, and cursors allow for the inclusion into the SAMEDL of DBMS-specific, that is, non-standard, operations and features, while preserving the benefits of standardization. These DBMS-specific extensions may be *verbs*, such as connect and disconnect, that signal the beginning and end of program execution, or *functions*, such as date manipulation routines, that extract the month from a date. The use of extensions, particularly the **extended** keyword, serves to mark those modules, tables, views, cursors, and procedures that go outside the standard and may require effort should the underlying DBMS be changed.

```

extended_schema_element ::=
    implementation defined
extended_table_element ::=
    implementation defined
extended_statement ::=
    implementation defined
extended_cursor_specification ::=
    implementation defined
extended_query_specification ::=
    implementation defined
extended_cursor_statement ::=
    implementation defined

```

Notice that the grammar is arranged such that extensions cannot influence the formation of

- procedure names
- cursor names
- module names
- table names
- view names
- status clauses, status parameter names and types, and standard post processing

Although the syntax and semantics of extensions are implementation defined, any portion of an extension whose semantics is expressible in standard SAMeDL shall be expressed in standard SAMeDL syntax. An extension may expand the class of:

- Value expressions by adding operators and functions. The operands of those operators and functions shall be restricted by rules similar to those in 6.4 and 8.10 of this specification.
- Search conditions by adding atomic predicates. Operands of those atomic predicates shall likewise be restricted according to rules such as those in 8.11 of this specification.
- Input parameter lists by adding the mode **out** to the parameter declarations, according to the rules of 8.6 of this specification.
- Table elements. If the extended table element is in the form of a column definition, the domain reference shall be present (see 7.2 of this specification).

Similarly, database data returned from extended procedures and cursors shall be defined in the syntax and semantics of select parameter lists (8.7 of this specification) with the syntax and semantics of the extended value expression class replacing the standard syntax and semantics. Such outputs shall be record objects. An extended statement returning such data shall accept an `into_clause` as described in 8.9 of this specification for specifying the record parameter name and type.

7 Data Description Language and Data Semantics

7.1 Definitional Modules

Definitional modules contain declarations of base domains, domains, subdomains, constants, records, enumerations, exceptions, and status maps. An Ada library unit package declaration is defined for each definitional module.

```

definitional_module ::=
    [context]
    [extended] definition module identifier_1 is
        {definition}
    end [identifier_2];

definition ::=
    base_domain_declaration |
    domain_declaration | subdomain_declaration |
    constant_declaration | record_declaration |
    enumeration_declaration | exception_declaration |
    status_map_declaration

```

1. When present, `identifier_2` shall be equivalent to `identifier_1`. (See 5.3 of this specification.)

Notes:

No `with_schema_clause` shall appear in the context of a definitional module. (See 6.2 of this specification.)

No two declarations within a definitional module shall have the same name, except for enumeration literals (see 7.1.6 of this specification).

End Notes.

Ada Semantics

For each definitional module within a compilation unit there is a corresponding Ada library unit package the name of which is the name of the definitional module, that is, AdaID(identifier_1). The Ada construct giving the Ada semantics of each definition within a definitional module is declared within the specification of that package. Nothing else appears in the specification of that package.

7.1.1 Base Domain Declarations

Base domains are the basis on which domains are defined. A base domain declaration has three parts: a sequence of parameters, used in domain declarations to supply information to the other two parts; a sequence of patterns, used to produce Ada source code in support of a domain; and a sequence of options, used by the compiler in implementation-defined ways.

```
base_domain_declaration ::=
  [extended] base domain identifier_1
  [(base_domain_parameter_list)]
  is
    patterns
    options
  end [identifier_2];

base_domain_parameter_list ::=
  base_domain_parameter {; base_domain_parameter}
```

1. If present, identifier_2 shall be equivalent to identifier_1. Identifier_1 is the name of the base domain.
2. The keyword **extended** may appear in a base_domain_declaration only if it also appears in the enclosing module declaration.

7.1.1.1 Base Domain Parameters

```
base_domain_parameter ::=
  identifier : data_class [ := static_expression]
  |
  map := pos
  |
  map := image

data_class ::=
  integer | fixed | float
  |
  character | bit
  |
  date | time | timestamp | interval
  |
  enumeration
```

1. The Ada identifiers within the list of base_domain_parameters of a base_domain_declaration are the names of the parameters that may appear in a parameter_association_list within a domain_declaration based on this base domain (see 7.1.3 of this specification). The static_expression within a base_domain_parameter, when present, specifies a default value for the parameter. This default value shall be of the correct data_class; that is, in the parameter declaration

```
Id : dcl := expr;
```

where *dcl* is a data_class, DATACLASS(expr) shall be *dcl*. Further, DATACLASS(Id) is *dcl*, whether or not the initializing expression *expr* is present, and DOMAIN(Id) is NO_DOMAIN.

2. A base domain is classified by its data_class. That is, an enumeration base domain is a base domain whose data_class is **enumeration**, a fixed base domain is a base domain whose data_class is **fixed**, etc. Integer, fixed and float base domains are collectively known as numeric base domains; date, time, timestamp and interval base domains are collectively known as

temporal base domains.

3. Every enumeration base domain has two predefined parameters: **enumeration** and **map**. These parameters are special in that the values that are assigned to them by a domain declaration (see 7.1.3 of this specification) are not of any of the data classes listed above. The value of an **enumeration** parameter is an enumeration_reference (see 6.3 of this specification); the value of **map** is a database_mapping (see 7.1.3 of this specification). A base domain declaration may explicitly declare a **map** parameter for the purpose of assigning a default mapping. An enumeration base domain shall not redefine the predefined base_domain_parameter **enumeration**.

There are two possible default mappings: **pos** and **image**. The value **pos** specifies that the Ada predefined attribute function 'POS' of the Ada type corresponding to the enumeration_reference, which is the **enumeration** parameter value in the domain declaration, shall be used to translate enumeration literals to their database encodings. Similarly for **image** and the 'IMAGE' attribute. See annex A of ISO/IEC 8652:1995 and 7.1.3 and 7.3 of this specification.

Every enumeration base domain whose **dbms type** is **char** or **character** shall have a third predefined parameter, **length** whose value is an integer of an implementation defined range. Such an enumeration base domain shall not redefine the predefined base_domain_parameter **length**.

4. Every fixed base domain has a predefined parameter **scale** whose value is an integer of an implementation defined range (see 6.1 of ISO/IEC 9075:1992). A fixed base domain shall not redefine the predefined base_domain_parameter **scale**.
5. Every character and bit base domain has a predefined parameter **length** whose value is an integer of an implementation defined range (6.1 of ISO/IEC 9075:1992). No character nor bit base domain shall redefine the predefined base_domain_parameter **length**.
6. Every interval base domain has a predefined parameter **qualifier** whose value shall be an SQL interval qualifier. See 10.1 of ISO/IEC 9075:1992 and 7.1.3 of this specification. No interval base domain shall redefine the predefined base_domain_parameter **qualifier**.

7.1.1.2 Base Domain Patterns

The patterns portion of a base domain declaration forms a template for the generation of Ada text, which forms the Ada semantics of domains based on the given base domain.

```

patterns ::= {pattern}
pattern ::=
  domain_pattern | subdomain_pattern | derived_domain_pattern
domain_pattern ::=
  domain pattern is pattern_list
  end pattern;
subdomain_pattern ::=
  subdomain pattern is pattern_list
  end pattern;
derived_domain_pattern ::=
  derived domain pattern is pattern_list
  end pattern;
pattern_list ::=
  pattern_element {pattern_element}
pattern_element ::= character_literal

```

VERSION 2

Patterns are used to create the Ada constructs that implement the Ada semantics of a domain, subdomain, or derived domain declaration (see 7.1.3 of this specification). Patterns are considered templates; parameters within a pattern are replaced by the values assigned to them either in the domain declaration, by inheritance, or by default. See 7.1.3 of this specification.

For a parameter to be recognized as such in a pattern, it is enclosed in square brackets ([,]). For the purpose of pattern substitution, a base domain may use a parameter **self**. When a pattern is instantiated, **self** is the result of applying the transformation AdalD to the name of the domain or subdomain being declared (see 5.3 of this specification). A base domain may use a parameter **parent** for the purpose of pattern substitution in a subdomain_pattern or a derived_domain_pattern. When such a pattern is instantiated, **parent** is AdalD applied to the name of the parent domain (see 5.3 and 7.1.3 of this specification).

Within a given character_literal of a pattern, a substring contained in matching curly brackets ({,}) is an optional phrase. Optional phrases may be nested. An optional phrase appears in the instantiated template if all parameters within the phrase have values assigned by a domain declaration (see 7.1.3 of this specification); the phrase does not appear when none of the parameters within the phrase has an assigned value. If some but not all parameters within an optional phrase have values assigned by a given domain declaration, the declaration is in error.

7.1.1.3 Base Domain Options

```
options ::= {option}
option ::= fundamental | for word_list use pattern_list ;
fundamental ::= classification_options | conversion_options
classification_options ::=
    for data class use data_class;
    for dbms_type use dbms_type [pattern_list];
    [for interface type use inter_type;]
conversion_options ::=
    for not null type name use pattern_list;
    for null type name use pattern_list;
    for conversion from type to type use converter ;
dbms_type ::=
    int | integer | smallint
    | real | double precision
    | char | character | bit
    | implementation defined
inter_type ::= dbms | none | dbms_type
type ::= dbms | not null | null
converter ::= function pattern_list | type mark
word_list ::= implementation defined
```

Options are used to define aspects of base domains that are essential to the declaration of domains within the SAMeDL. Implementations may define options beyond the fundamental options. The classification options are required in all base domains. If the **interface type** option is missing, the option

```
for interface type use dbms
```

is assumed. The conversion options are required if the interface type is not **none**. The meanings of the fundamental options are given by the following list.

1. The **interface type** of a base domain is the *SQL_data_type* (see 6.1 of ISO/IEC 9075:1992) to be used when declaring parameters of the concrete interface (SQL module) for all objects of domains based directly or indirectly on the base domain. See 8.6, 8.7, and 8.8 of this specification.

If the dbms type of a base domain is *DT*, an interface type specification of **dbms** is equivalent to the specification

```
for interface type use DT;
```

If an interface type of **none** is specified, then domains based directly or indirectly on the base domain shall not be used for the parameters of any procedure.

Note: Objects of temporal data types can not appear as parameters to SQL procedures.

If the interface type of a base domain is implementation defined, the keyword **extended** shall appear in the declaration of the base domain.

2. The **dbms type** of a base domain is the *SQL_data_type* (see 6.1 of ISO/IEC 9075:1992) to be used when declaring columns having a domain based directly or indirectly on the base domain and when using the domain as the target of an SQL CAST expression. See 7.2 and 8.10 of this specification.

If the dbms type of a base domain is implementation defined, the keyword **extended** shall appear in the declaration of the base domain.

3. The **null** and **not null** type names are the targets of the function AdaTYPE. They are the names of the types of parameters and parameter components in Ada procedures. See 8.6 and 8.7 of this specification.
4. The **data class** option specifies the data class (see 7.1.1.1 of this specification) of all objects of any domain based on this base domain. If *BD* is a base domain to which the data class *dc* is assigned by an option in its definition, and if *D* is a domain based directly or indirectly (see 7.1.3 of this specification) on *BD*, then DATACLASS(*D*)=*dc*. The data class governs the use of literals with such objects (see 8.8 and 8.10 of this specification).
5. An operand of the **conversion** option is a means of converting non-null data between objects of the not null-bearing type, the null-bearing type (see 7.1.3 of this specification) and the dbms type associated with a domain. A method shall be a function, an attribute of a type, or a type conversion. A means of determining the identity of these methods shall appear in the options of a base domain. The identity of a method may be given as a pattern containing parameters.

However, enumeration domains do not have converters between the dbms type and the not null-bearing type, as the **map** parameter predefined for all enumeration domains describes a conversion method between enumeration and database representations of non-null data. The method is the application, as appropriate, of the function described by the database_mapping that is the operand of the map parameter association. See 7.1.1.1 and 7.1.3 of this specification.

7.1.2 The SAME Standard Base Domains

The predefined definitional module, SAMeDL_Standard, contains the declarations of the predefined SAME Standard Base Domains: SQL_Int, SQL_Smallint, SQL_Char, SQL_Nchar, SQL_VarChar, SQL_Bit, SQL_Real, SQL_Double_Precision, SQL_Enumeration_as_Char, and SQL_Enumeration_as_Int. The text of SAMeDL_Standard appears in Annex A of this specification.

7.1.3 Domain and Subdomain Declarations

```

domain_declaration ::=
    domain identifier is new bas_dom_ref [not null]
    [ ( parameter_association_list ) ] ;

subdomain_declaration ::=
    subdomain identifier is dom_ref [not null]
    [ ( parameter_association_list ) ] ;

dom_ref ::= domain_reference | subdomain_reference
bas_dom_ref ::= dom_ref | base_domain_reference

parameter_association_list ::=
    parameter_association {, parameter_association}

parameter_association ::=
    identifier => static_expression
    | map => database_mapping
    | enumeration => enumeration_reference
    | scale => static_expression |
    | length => static_expression
    | qualifier => SQL_interval_qualifier

database_mapping ::=
    enumeration_association_list | pos | image

enumeration_association_list ::=
    ( enumeration_association {, enumeration_association} )

enumeration_association ::=
    enumeration_literal => database_literal

```

1. Consider the domain declaration:

domain DD is new EE ...

- a. If EE is a base_domain_reference, then EE is said to be the base domain of DD.
- b. Otherwise, EE is a domain_reference or subdomain_reference, the base domain of DD is defined to be the base domain of EE, DD is said to be derived from EE, and EE is said to be the parent of DD.

2. Similarly, in the subdomain declaration

subdomain FF is GG ...

the base domain of FF is defined as the base domain of GG, FF is said to be a subdomain of GG, and GG is said to be the parent of FF.

3. The database type of a domain D , denoted as $\text{DBMS_TYPE}(D)$, is the value, appropriately parameterized, of the **for dbms type** option from the base domain of D . See 7.1.1.3 of this specification.
4. The interface type of a domain D , denoted as $\text{INTER_TYPE}(D)$, the value, appropriately parameterized, of the **for interface type** option from the base domain of D . See 7.1.1.3 of this specification.
5. If $\text{INTER_TYPE}(D)$ is not **none**, the values of $\text{DBMS_TYPE}(D)$ and $\text{INTER_TYPE}(D)$ shall be such that conversion between them is legal in both directions by the rules of 9.1 and 9.2 of ISO/IEC 9075:1992.
6. The data class of a domain D , denoted $\text{DATACLASS}(D)$, is the data class of its base domain, the value of the **for data class** option. A domain is numeric if its data class is numeric (see 7.1.1.1 of this specification).
7. Except for **scale**, **enumeration**, **length**, **map** and **qualifier**, an identifier within a parameter_association shall be the name of a base_domain_parameter in the declaration of the base do-

main of the domain or subdomain being declared. See 7.1.1.1 of this specification.

8. A domain or subdomain D is said to *assign the expression E* to the parameter P , if
 - a. the parameter_association $P \Rightarrow E$ appears in the declaration of D ; or
 - b. (a) does not hold, D is a subdomain or a derived domain, and the parent domain assigns the expression E to the parameter P ; or
 - c. (a) and (b) do not hold and in the base_domain_declaration for the base domain of D , the base_domain_parameter

$P : class := E$

appears.

In all cases, $DATACLASS(E)$ shall be $DATACLASS(P)$ as defined by the declaration of the base domain. See 7.1.1.1 of this specification.

9. If a domain D assigns the expression E to a parameter P , then
 - $DOMAIN(D.P) = NO_DOMAIN$.
 - $DATACLASS(D.P) = DATACLASS(E)$
10. A domain_declaration shall assign an expression to each base_domain_parameter that appears in any non-optional phrase
 - of the base domain's domain_pattern, if the declaration is not declaring a derived domain;
 - of the base domain's derived_domain_pattern, if the declaration is the declaration of a derived domain.

Similar rules govern subdomain_declarations and subdomain_patterns. See 7.1.1.2 of this specification.

11. The value assigned to the **scale** parameter in the declaration of a fixed domain shall be an integer from an implementation defined range.
12. The value assigned to the **length** parameter in the declaration of a character domain shall be an integer from an implementation defined range.
13. The value assigned to the **qualifier** parameter in the declaration of an interval domain shall be an *SQL_interval_qualifier*. The value of the qualifier parameter of an interval domain D is denoted $QUALIFIER(D)$.
14. Any domain_declaration or subdomain_declaration of an enumeration domain shall assign an enumeration_reference as to the base_domain_parameter **enumeration** and a data-base_mapping to the base_domain_parameter **map**. If the **map** parameter is assigned an enumeration_association_list, then
 - a. Each enumeration_literal within the enumeration referenced by the enumeration_reference given by the **enumeration** parameter shall appear as the enumeration_literal of exactly one enumeration_association.
 - b. No database_literal shall appear in more than one enumeration_association.

Note: These constraints ensure that the database_mapping is an invertible (i.e., one-to-one) function. That function is used for both compile time and runtime data conversions. See 7.1.1.3 and 7.3 of this specification.

Let D be an enumeration domain or subdomain declaration and let En be the name of the enumeration referenced by the value assigned by D to the **enumeration** base_domain_parameter. D is said to *assign the expression E* to the enumeration literal Ei , if D assigns the data-base_mapping M as the value of the **map** base_domain_parameter and M

- is **pos**, and $E = En'Pos(Ei)$, or

VERSION 2

- is **image**, and $E = \text{EnImage}(E)$, or
- is an enumeration_association_list containing an enumeration_association of the form $E/ \Rightarrow E$

See 7.1.6 of this specification.

15. The database_mapping of an enumeration domain or subdomain declaration D should preserve the ordering implied by that domain's enumeration_reference ER . That is, if L_1 and L_2 are enumeration literals of ER such that L_1 occurs before L_2 in ER 's enumeration_literal_list, then the value assigned to L_1 by D should be less than the value assigned to L_2 by D .
16. A domain or subdomain is said to be *not null only* if it or any of its parent domains is declared with the **not null** phrase. In that case no object of the domain can contain the null value.

Ada Semantics

An instantiation of a pattern defined for the base domain of the domain being declared, as described in 7.1.1.2 of this specification, shall appear within the Ada package specification corresponding to the module within which the domain_declaration appears. If in the domain declaration:

```
domain DD is new EE ...
```

EE is a base_domain_reference, then the domain_pattern is instantiated; if EE is a domain_reference, the derived_domain_pattern is instantiated; for the subdomain declaration

```
subdomain FF is GG ...
```

the subdomain_pattern is used.

Note: Examples

The following examples illustrate the declaration of domains and have been annotated with references to the appropriate clauses of the language definition. The base domains used in these examples exist in the predefined definitional module SAMeDL_Standard, which appears in Annex A of this specification. The constant Max_SQL_Int is declared in the predefined definitional module SAMeDL_System. Both SAMeDL_Standard and SAMeDL_System (see Annex B of this specification) are assumed to be visible, as is the enumeration declaration *Colors* (see 7.1.6 of this specification).

```
domain SWITCHES is new SQL_Bit (Length => 4);

domain Weight is new SQL_Int ( -- 7.1.3: #1a
    First => 0,                -- 7.1.3: #5 and #8
    Last  => Max_SQL_Int);     -- 7.1.3: #5 and #8

domain Weight_In_Pounds is new Weight;
domain Weight_In_Grams  is new Weight;

domain City_Names is new SQL_Char ( -- 7.1.3: #1a
    Length => 15);              -- 7.1.3: #5 and #10

domain Colors is new SQL_Enumeration_As_Char ( -- 7.1.3: #1a
    enumeration => Color_Values,              -- 7.1.3: #11
    map         => image);                    -- 7.1.3: #11 and #12

domain Auto_Weight is new Weight ( -- 7.1.3: #1b
    Last => 10000);                -- 7.1.3: #5

subdomain Auto_Part_Weight is Auto_Weight ( -- 7.1.3: #2
    Last => 2000);                -- 7.1.3: #5 and #8

domain DB_Date is new SQL_Date;

domain DB_Date_Strings is new SQL_VarChar;
```

These declarations produce the following Ada code.

```
-- the instantiation of an SQL_Bit domain
type SWITCHESNN_Base is new SQL_Bit_Not_Null;
```

VERSION 2

```
subtype SWITCHES_Not_Null is SWITCHESNN_Base (1 .. 4);
type SWITCHES_Base is new SQL_Bit;
subtype SWITCHES_Type is SWITCHES_Base (SWITCHES_Not_Null'Length);
package SWITCHES_Ops is new SQL_Bit_Ops(
    SWITCHES_Base, SWITCHESNN_Base);

-- the Ada code below is the instantiation of the domain pattern
-- from base domain SQL_Int

type Weight_Not_Null is new SQL_Int_Not_Null
    range 0 .. implementation_defined;
type Weight_Type is new SQL_Int;
package Weight_Ops is new SQL_Int_Ops (
    Weight_Type, Weight_Not_Null);

-- The domains Weight_In_Pounds and Weight_In_Grams are not illustrated
-- here as there instantiations are essentially identical to Weight.

-- the Ada code below is the instantiation of the domain pattern
-- from base domain SQL_Char

type City_NamesNN_Base is new SQL_Char_Not_Null;
subtype City_Names_Not_Null is City_NamesNN_Base (1 .. 15);
type City_Names_Base is new SQL_Char;
subtype City_Names_Type is City_Names_Base (City_Names_Not_Null'Length);
package City_Names_Ops is new SQL_Char_Ops(
    City_Names_Base, City_NamesNN_Base);

-- the Ada code below is the instantiation of the domain pattern
-- from the base domain SQL_Enumeration_As_Char

package Colors_Pkg is new SQL_Enumeration_Pkg(Color_Values);
type Colors_Type is new Colors_Pkg.SQL_Enumeration;

-- the Ada code below is the instantiation of the derived domain
-- pattern from the base domain SQL_Int

type Auto_Weight_Not_Null is new Weight_Not_Null
    range Weight_Not_Null'First .. 10000;
type Auto_Weight_Type is new Weight_Type;
package Auto_Weight_Ops is new SQL_Int_Ops(
    Auto_Weight_Type, Auto_Weight_Not_Null);

-- the Ada code below is the instantiation of the subdomain
-- pattern from the base domain SQL_Int

subtype Auto_Part_Weight_Not_Null is Auto_Weight_Not_Null
    range Auto_Weight_Not_Null'First .. 2000;
type Auto_Part_Weight_Type is new Auto_Weight_Type;
package Auto_Part_Weight_Ops is new SQL_Int_Ops(
    Auto_Part_Weight_Type, Auto_Part_Weight_Not_Null);

-- the DB_Date base domain generates no Ada code

-- DB_DATE_String is an example of SQL_VarChar
type DB_Date_Strings_Not_Null is new SQL_VarChar_Not_Null;
type DB_Date_Strings_Type is new SQL_VarChar;
package DB_Date_Strings_Ops is new SQL_VarChar_Ops(
    DB_Date_Strings_Type, DB_Date_Strings_Not_Null);
```

End Examples

7.1.4 Constant Declarations

```
constant_declaration ::=
    constant identifier [: domain_reference]
        is static_expression ;

static_expression ::= value_expression
```

A static expression is a value expression (see 8.10 of this specification) whose value can be calculated at compile time; i.e., whose leaves are all either literals or constants.

Now let K denote the constant declaration

```
constant C [: D ] is E ;
```

VERSION 2

1. DATACLASS(*K*) is DATACLASS(*E*), the data class of the expression *E*. See 5.4 and 8.10 of this specification.
2. If DATACLASS(*K*) is **enumeration**, **character** or **bit** then *D* shall be present.
3. If the domain_reference *D* is not present, then
 - a. *C* is a *universal* constant of class DATACLASS(*K*).
 - b. If DATACLASS(*K*) is not a temporal data class, then AdaTYPE(*K*) is an anonymous type, *universal_T*, where *T* is DATACLASS(*K*).
 - c. DOMAIN(*K*) = NO_DOMAIN
4. If the domain_reference *D* is present, then
 - a. DOMAIN(*K*)=*D* and *E* shall conform to *D* (see 6.4 of this specification).
 - b. If DATACLASS(*K*) is not a temporal data class, then AdaTYPE(*K*) is defined as the type name within *D* designated as not null-bearing. That type name shall not refer to a limited type.

Ada Semantics

Let VALUE represent the function which calculates the value of a static_expression. Let *SE* be a static_expression. VALUE(*SE*) is given recursively as follows:

1. If *SE* contains no operators, then
 - a. If *SE* is a database_literal, then VALUE(*SE*) = *SE*.
 - b. If *SE* is an enumeration_literal of domain *D*, and *D* assigns expression *E* to that enumeration literal, then VALUE(*SE*) = *E*.
 - c. If *SE* is a reference to the constant whose declaration is given by

```
constant C [: D ] is E ;
```

then VALUE(*SE*) = VALUE(*E*).
 - d. If *SE* is a reference to a parameter *P* from domain *D*, and *D* assigns the expression *E* to *P*, then VALUE(*SE*) = VALUE(*E*).
2. If *SE* is *D*(*SE*₁), where *D* is a domain name, then VALUE(*D*(*SE*₁)) = VALUE(*SE*₁).
3. If *SE* is +*SE*₁ (or -*SE*₁) then VALUE(*SE*) = +VALUE(*SE*₁) (or -VALUE(*SE*₁)).
4. If *SE* is *SE*₁ *op* *SE*₂, where *op* is an arithmetic operator, then VALUE(*SE*) = VALUE(*SE*₁) *op* VALUE(*SE*₂) where *op* is evaluated according to the rules of SQL. See 6.12 of ISO/IEC 9075:1992.
5. If *SE* is (*SE*₁) then VALUE(*SE*) = (VALUE(*SE*₁)).

If DATACLASS(*K*) is not a temporal data class, let *K* denote the constant declaration

```
constant C [: D ] is E ;
```

If *E* is not an enumeration literal, let *Q* be the Ada representation of VALUE(*E*); otherwise, let *Q* be AdaID(*E*). Then the Ada library unit package specification corresponding to the module in which the constant declaration *K* appears shall have an Ada constant declaration of the form

```
AdaID(C) : constant [AdaTYPE(K)] := Q ;
```

The type designator AdaTYPE(*K*) is omitted from this declaration if it is an anonymous type.

Note: Temporal constants have no direct Ada analogue and therefore do not generate Ada constants.

VERSION 2

Note: Examples

The following are examples of constant declarations.

```
constant Grams_In_Pound is 453.59237;
constant The_Color_Red : Colors is Red;
constant Home_Port : City_Names is 'Pittsburgh';
constant All_On : SWITCHES is B'1111'
```

Note that Grams_in_Pound is a universal constant of class **fixed**. These declarations generate the following Ada declarations.

```
Grams_In_Pound : constant := 453.59237;
The_Color_Red : constant : Color_Values := Red;
Home_Port : constant City_Name_Not_Null := "Pittsburgh";
All_On: constant SWITCHES:= (1,1,1,1);
```

End Examples

7.1.5 Record Declarations

```
record_declaration ::=
    record identifier_1 [named_phrase] is
        component_declarations
    end [identifier_2] ;
named_phrase ::= named identifier
component_declarations ::= component_declaration {component_declaration}
component_declaration ::=
    component {, component} : domain_reference [not null] ;
component ::= component_name [dblength [named_phrase]]
component_name ::= Ada_identifier
```

If present, identifier_2 shall be equivalent to identifier_1. Identifier_1 is the *name* of the record.

Let R be a record declaration. Define AdaNAME(R) to be

1. The alias N , if the named_phrase **named** N appears in the declaration.
2. Row , otherwise.

Note: AdaNAME(R) is the default for the name of the row record formal parameter in the parameter profile of any procedure that uses the declaration R . See 8.2, 8.5, and 8.9 of this specification.

Ada Semantics

The Ada library unit package specification corresponding to the module within which the record_declaration R appears shall have an Ada record type declaration R_{Ada} defined as follows:

1. The name of the record type R_{Ada} shall be AdaID(identifier_1).
2. For some integer k , let the component_declarations of R be given by the sequence

components _{i} : D _{i} [**not null** _{i}]
for $1 \leq i \leq k$, where components _{i} is given by the sequence

C _{ij} [**dblength** _{ij} [**named** N _{ij}]]
where $1 \leq j \leq m_i$ for some integer m_i , R_{Ada} shall be equivalent, in the sense of 3.2.10 and 3.7.2 of ISO/IEC 8652:1995, to a record type whose components are given by the sequence

COMP _{ij} [DBleng _{ij}]

where i and j are bound as before and $COMP_{ij}$ is given by

$AdaID(C_{ij}) : T_i ;$

where T_i is an Ada type name determined to be:

- a. The not null-bearing type name within the domain D_i , if either D_i is a not null only domain or **not null** _{i} is present in R_i ;
- b. Otherwise the null-bearing type name within the domain D_i .

The optional component $DBLeng_{ij}$ appears if the optional **dblength** _{ij} phrase appears and in that case is of the form

$DBLNgNAME_{ij} : Ada_Indicator_Type ;$

where $DBLNgNAME_{ij}$ is $AdaID(N_{ij})$ if N_{ij} appears and is $AdaID(C_{ij})_DbLength$, otherwise; and $Ada_Indicator_Type$ is the type `SQL_Standard.Indicator_Type` (see 12.3.8.a.iii of ISO/IEC 9075:1992).

Note: Examples

The following example illustrates the declaration of a record object.

```
record Parts_Row_Record_Type named Parts_Row_Record is
  Part_Number      : Part_Numbers not null;
  Part_Name        : Part_Names;
  Color            : Colors;
  Weight           : Weight_In_Grams;
  City             : City_Names;
end Parts_Row_Record_Type;
```

This declaration produces the following Ada code. It has been annotated with references to the appropriate clauses of the language definition.

```
type Parts_Row_Record_Type is record -- Ada Semantics #1
  Part_Number      : Part_Numbers_Not_Null; -- Ada Semantics #2
  Part_Name        : Part_Names_Type;
  Color            : Colors_Type;
  Weight           : Weight_In_Grams_Type;
  City             : City_Names_Type;
end record;
```

End Examples

7.1.6 Enumeration Declarations

Enumerations are used to declare sets of enumeration literals for use in enumeration domains and status maps. See 7.1.3 and 7.1.8 of this specification.

```
enumeration_declaration ::=
  enumeration identifier_1 is ( enumeration_literal_list ) ;
enumeration_literal_list ::=
  enumeration_literal { , enumeration_literal }
```

1. Identifier_1 is the *name* of the enumeration.
2. Each identifier within an enumeration_literal_list is said to be an enumeration literal of the enumeration. The enumeration_declaration is considered to declare each of its enumeration_literals. An enumeration_literal may appear in multiple enumeration_declarations.

Ada Semantics

There shall be, within the Ada package specification corresponding to the module within which an enumeration appears, a type declaration of the form

```
type AdaID(identifier_1) is ( AdaID(enumeration_literal_list) ) ;
```

The application of the transformer AdaID (see 5.3 of this specification) to a list is accomplished by application of the transformer to each element of the list. Ada character literals shall not be used in enumerations. See 5.4 of this specification.

Note: Examples

The following are examples of enumeration declarations.

```
enumeration Single_Row_Status is (
    More_Than_One_Row, No_Such_Row, Row_Found);

enumeration Color_Values is (
    Purple, Blue, Green, Yellow, Orange, Red, Black, White);
```

The above declarations produce the following Ada code.

```
type Single_Row_Status is (
    More_Than_One_Row, No_Such_Row, Row_Found);

type Color_Values is (
    Purple, Blue, Green, Yellow, Orange, Red, Black, White);
```

End Examples

7.1.7 Exception Declarations

```
exception_declaration ::= exception identifier ;
```

Identifier is the *name* of the exception.

Ada Semantics

There shall be, within the Ada package specification corresponding to the module within which an exception declaration appears, an exception declaration of the form

```
AdaID(identifier_1) : exception ;
```

Note: Examples

The following are examples of exception declarations.

```
exception Data_Definition_Does_Not_Exist;
exception Insufficient_Privilege;
exception Transaction_Rollback;
```

The above declarations produce the following Ada code.

```
Data_Definition_Does_Not_Exist : exception;
Insufficient_Privilege : exception;
Transaction_Rollback : exception;
```

End Examples

7.1.8 Status Map Declarations

The execution of any procedure (see 6.6, 8.2, and 8.5 of this specification) causes the execution of an SQL procedure. That execution causes a special parameter, called the SQL status parameter, to be “set to a status code that either indicates that a call of the procedure completed successfully or that an exception condition occurred during execution of the procedure” (See 4.18.1 of ISO/IEC 9075:1992). Status maps are used within abstract modules to process the status data in a uniform way. Each map declares a partial function from the set of all possible SQL status parameter values onto (1) enumeration literals of an enumeration and (2) raise statements.

```

status_map_declaration ::=
  [ sqlcode | sqlstate ] status identifier_1
  [named_phrase]
  [uses target_enumeration]
  is ( status_assignment { , status_assignment } );

target_enumeration ::=
  enumeration_reference | boolean

status_assignment ::=
  left_hand_side => enumeration_literal
  | left_hand_side => raise exception_reference

left_hand_side ::= static_expr { , static_expr }

static_expr ::=
  static_expression
  | static_expression .. static_expression
  (see 3.5 of ISO/IEC 8652:1995)

```

1. Identifier_1 is the *name* of the map.
2. A target_enumeration of **boolean** is a reference to the predefined Ada enumeration type Standard.boolean.
3. If the optional **uses** clause is not present, then only status_assignments that contain **raise** shall be present in the status_map_declaration.
4. Every Ada_enumeration_literal within a status_assignment shall be an Ada_enumeration_literal within the enumeration referenced by the target_enumeration.
5. If neither **sqlcode** nor **sqlstate** is specified, then **sqlstate** is assumed.
6. If **sqlcode** is specified, then if E is a static_expression appearing in a left_hand_side, then DATACLASS(E) = **integer**. In this case a static_expr of the form E₁ .. E₂ shall be permitted in a left_hand_side, provided E₁ ≤ E₂. If v₁, v₂, ..., v_m are all the integers between E₁ and E₂ inclusive, then the form E₁ .. E₂ is equivalent to the list of static expressions v₁, v₂, ..., v_m.
7. If **sqlstate** is specified, then if E is a static_expression appearing in a left_hand_side, then DATACLASS(E) = **character**, and either
 - a. E shall contain five characters and, if DOMAIN(E) ≠ NO_DOMAIN, then DOMAIN(E) = SAMeDL_Standard.SQLSTATE_Domain; or
 - b. E shall contain two characters and, if DOMAIN(E) ≠ NO_DOMAIN, then DOMAIN(E) = SAMeDL_Standard.SQLSTATE_Class_Domain. If v₁, v₂, ..., v_m are all the SQLSTATE values whose Class value is given by the value of E, then E is equivalent to the list of static expressions v₁, v₂, ..., v_m. See annex A for the text of the predefined module SAMeDL_Standard. See 22.1 of ISO/IEC 9075:1992 for the standard values of SQLSTATE. Other values of SQLSTATE may be defined by an implementation.
8. If E and E' are distinct static expressions appearing in the clauses of a status map (possibly via

VERSION 2

the equivalence of previous rules), then E and E' shall not evaluate to the same value.

Notes: **sqlstate** is the preferred form. **sqlcode** is a feature deprecated in ISO/IEC 9075:1992, annex D.

A status_assignment takes the form of a list of alternatives as found in Ada case statements, aggregates, and representation clauses. The **others** choice is not valid for status_assignments, however.

SAMeDL_Standard contains the definition of a status map Standard_Map, defined as follows:

```
sqlstate status Standard_Map
  named Is_Found
  uses boolean
  is
    (Successful_Completion_No_Subclass => True,
     No_Data_No_Subclass => False);
```

Standard_Map is the status map for those fetch statements that appear in cursor declarations by default. (See 8.5 of this specification.) It signals end of table by returning false.

Note: Examples

The following is an example of a status map declaration. For the enumeration and exception declarations, refer to the examples in 7.1.6 and 7.1.7 of this specification. They are assumed to be visible at the point at which the status map is declared. Further, direct visibility (i.e., **use**) of SAMeDL_Standard is assumed.

This map might be usefully applied to single row select statements. It distinguishes the exceptional condition of retrieving more than one row from the condition of retrieving no rows. It eliminates the distinction between a successful completion and a completion in which a warning was returned. It raises an exception in the Ada application in two subcases of an SQL transaction rollback. See 6.5 and Annex A of this specification and 22 in ISO/IEC 9075:1992.)

```
status Single_Row_Select_Map
  named Result
  uses Single_Row_Status is (
    Cardinality_Violation_No_Subclass => More_Than_One_Row,
    No_Data_No_Subclass               => No_Such_Row,
    Successful_Completion_No_Subclass,
    Warning                           => Row_Found,
    Transaction_Rollback_No_Subclass,
    Transaction_Rollback_Serialization_Failure => raise Transaction_Rollback);
```

End Examples

7.2 Schema Modules

```
schema_module ::=
  [context]
  [extended] schema module identifier_1 is
    {schema_element}
  end [identifier_2];

schema_element ::=
  schema_procedure
  | schema_definition_statement

schema_procedure ::=
  [extended] procedure identifier_3 is
    schema_definition_statement
    [status_clause]
  end [identifier_4];

schema_definition_statement ::= see the rules
```

1. If present, identifier_2 shall be equivalent to identifier_1. Identifier_1 is the *name* of the schema_module. Similarly, if present identifier_4 shall be equivalent to identifier_3. Identifier_3 is the *name* of the schema procedure.

VERSION 2

2. Identifier_1 shall be different from any other schema module name (See clause 11.1 of ISO/IEC 9075:1992). Identifier_3 shall be different from any other procedure name in the enclosing schema_module.
3. An extended_schema_element may appear in a schema_module only if the keyword **extended** appears in the associated schema module declaration.

A schema_definition_statement differs from schema definition statement as defined in ISO 9075:1992 in the following ways:

1. The keyword **create** is optional in a table and view definition. A view or table definition ends with the phrase **end** [T] ; where T is the name of the table or view being defined.

Note: In the prior version of this specification, the **create** keyword was not present in view and table definitions.

2. In all schema definitions statements:
 - a. Value expressions as defined in 8.10 replace SQL value expressions.
 - b. Search conditions as defined in 8.11 replace SQL search conditions.

Note: Conditions in ISO 9075:1992 prohibit value expressions in schema definition statements from containing input_references. Note that the select parameters in a query expression defining a view are those of SQL, not those in 8.7 of this specification.

3. In table definitions:
 - a. A column definition shall contain a domain_reference appearing in the syntactic location of a data type. A domain name, as defined in ISO 9075:1992 shall not appear in a column definition.
 - b. Suppose that column_definition *CD* contain a reference to the domain *D*.
 - i. The domain of *CD*, denoted DOMAIN(*CD*), is *D*.
 - ii. DATACLASS(*CD*) is DATACLASS(*D*).
4. In view definitions:
 - a. The i^{th} column of the defined view has domain DOMAIN_{*i*} as defined by the query expression contained in the view definition; see 8.4 of this specification.
 - b. DATACLASS of the i^{th} column of the defined view is DATACLASS(DOMAIN_{*i*}).

Ada Semantics

For each schema module within a compilation unit there is a corresponding Ada library unit package the name of which is the name of the schema module, that is AdaID(identifier_1).

For each schema procedure defined within a schema module, there is an Ada procedure defined within the corresponding Ada library unit package. The name of the Ada procedure is the name of the schema procedure, that is, AdaID(identifier_3). If a status_clause appears in the schema procedure, the Ada procedure will have a single parameter of mode **out**. For the name and type of this parameter, see 7.1.8 and 8.12 of this specification.

VERSION 2

SQL Semantics

There is an SQL module associated with each schema module. The name of the SQL module is implementation defined. The language clause of the SQL module shall specify Ada.

For each schema procedure within a schema module, there is an SQL procedure defined within the corresponding SQL module. The name of this procedure is implementation defined. The procedure declaration shall declare an SQLSTATE parameter unless a status_clause appears in the schema procedure that references a status map specifying **sqlcode**, in which an SQLCODE parameter is declared. (See 7.1.8 and 8.12 of this specification.)

The SQL schema definition statement within the SQL procedure is derived from the schema definition statement as follows:

1. Any search condition SC is replaced with SQL_{SC} (SC), defined in 8.11 of this specification. Any value expression VE is replaced with SQL_{VE} (VE) as defined in 8.10 of this specification.
2. For view and table definitions: if needed, the keyword **create** is prepended; the **end** [T] ; is removed.
3. For table definitions, the domain name, D, is replaced with DBMS_TYPE(D).
4. Otherwise, the statement is unchanged.

Interface Semantics

A call to the Ada procedure corresponding to a schema procedure shall have the effect of executing the corresponding SQL procedure and performing standard post processing as defined in 6.5 of this specification.

Note: Examples

The following is an example of a schema module.

```
schema module Parts_Suppliers_Database is
  -- the Parts table
  table P is      -- 7.2.1: #1 and #2
    PNO not null : Part_Numbers, -- 7.2.1: #3 and #4
    PNAME       : Part_Names,    -- 7.2.1: #5
    COLOR       : Colors,
    WEIGHT      : Weight_In_Pounds,
    CITY        : City_Names,
    unique (PNO)
  end P;
  -- the Suppliers table
  table S is      -- 7.2.1: # 1 and #2
    SNO not null : Supplier_Numbers, -- 7.2.1: #3 and #4
    SNAME       : Supplier_Names,    -- 7.2.1: #5
    "STATUS"    : Status_Values,     -- 5.3
    CITY        : City_Names,
    unique (SNO)
```

VERSION 2

```
end S;

-- the Orders table
table SP is          -- 7.2.1: #1 and #2
    SNO character(5) not null : Supplier_Numbers,          -- 7.2.1: #3 and #4
    PNO character(6) not null : Part_Numbers,
    QTY integer      : Quantities, -- 7.2.1: #5
    Date_Order : DB_Date
    unique (SNO, PNO)
end SP;

-- the Part_Number_City view
procedure Create_PNO_City_View is
    view PNO_CITY as -- 7.2.2: #1 and #2
        select distinct PNO, CITY
        from SP, S
        where SP.SNO = S.SNO
    end PNO_CITY;
;

end Parts_Suppliers_Database;
```

This schema module generates the following Ada package

```
package Parts_Supplier_Database is
    procedure Create_PNO_City_View;
```

and the following SQL procedure (within a module whose name is not specified by the standard.).

```
procedure implementation_defined (SQLSTATE)
    create view PNO_CITY as -- 7.2.2: #1 and #2
        select distinct PNO, CITY
        from SP, S
        where SP.SNO = S.SNO
;
;
```

End Examples

7.3 Data Conversions

The procedures that are described in an abstract module (see clause 8 of this specification) transmit data between an Ada application and a DBMS. Those data undergo a conversion during the execution of those procedures. Constants and enumeration literals used in statements are replaced by their database representation in the form of the statement in the concrete module. This process occurs at module compile time. Both processes are described in this clause.

Execution Time Conversions

The execution time conversions check for and appropriately translate null values; for not null values, the conversion method identified by the appropriate base domain definition (see 7.1.1.3 of this specification) is executed.

Input parameter conversion rule. If the type of an input parameter is null-bearing, then in the corresponding SQL procedure there is an associated *SQL_parameter_specification* to which an *SQL_indicator_parameter* has been assigned. (See 8.6 and 8.8 of this specification.) If, for any execution of the procedure, the value of the input parameter is null, then the indicator parameter is assigned a negative value. (See 4.17.2 and General Rule 6 in 6.2 of ISO/IEC 9075:1992.) Otherwise, the indicator parameter shall be non-negative and the SQL parameter shall be set from the input parameter by the conversion process identified for the

VERSION 2

base domain. If the type of an input parameter is not null-bearing, the SQL parameter shall be set from the input parameter by the conversion process identified for the base domain (7.1.1.3 of this specification).

Output parameter conversion rule. For output parameters of procedures containing either **fetch** or **select** statements, this process is run in reverse. Let SP be a select parameter. Then the corresponding SQL procedure has a data parameter and an indicator parameter corresponding to SP (see 8.2, 8.5, and 8.7 of this specification). For any execution of the procedure:

- If the indicator parameter is negative, then
 - if the type of the Ada record component $COMP_{Ada}(SP)$ (see 8.2 and 8.5 of this specification) is null-bearing, then $COMP_{Ada}(SP)$ is set to the null value; *else*
 - if the type of $COMP_{Ada}(SP)$ is not null-bearing, the exception `SAMeDL_Standard.Null_Value_Error` is raised.
- If the indicator parameter is non-negative, then the value of $COMP_{Ada}(SP)$ is set from the value of the SQL data parameter by the conversion process identified for the base domain of SP (see 7.1.1.3) of this specification). If the record component $DBLeng_{Ada}(SP)$ is present (see 8.2 and 8.4 of this specification), then it is set to the value of the indicator parameter.

Compile Time Conversions

The SQL semantics of constants, domain parameters, and enumeration literals (and constants that evaluate to enumeration literals) used in value lists of insert statements (see 8.8) and value expressions (see 8.10 of this specification) require that they be replaced in the generated SQL code by representations known to the DBMS. For enumeration literals, the enumeration mapping is used (see 7.1.1.1, 7.1.1.3 and 7.1.3 of this specification).

Let *V* be an identifier. If *V* is not a reference to a constant, a domain parameter, or an enumeration literal, then *V* is not static and undergoes no compile time conversion.

If *V* is a reference to

- a constant declared by

```
constant C [: D ] is E ;
```

- a domain parameter *param* of domain *D*, and *D* assigns the expression *E* to *param* (see 7.1.3 of this specification),
- or an enumeration literal *E* from enumeration domain *D* (see 8.3, 8.8, 8.10, and 8.11 of this specification), and *D* assigns the expression *E* to *V*,

then *V* is replaced by the static expression $SQL_{VE}(E)$ (see 8.10 of this specification).

8 Abstract Module Description Language

8.1 Abstract Modules

```
abstract_module ::=  
  [context]  
  [extended] abstract module identifier_1 is
```

```

SQL_module_authorization_clause]
{definition}
{temporary_table_declaration}
{module_contents}
end [identifier_2];
module_contents ::=
  cursor_declaration |
  dynamic_declare_cursor
  procedure_declaration

```

1. When present, identifier_2 shall be equivalent to identifier_1. Identifier_1 is the *name* of the abstract module.
2. No two of the items (that is, temporary table, procedures, cursors, and definitions) declared within an abstract module shall have the same name.
3. A temporary_table_declaration differs from an SQL temporary table declaration (see 13.11 of ISO 9075:1992) in the same way in which table declarations differ from SQL table definitions (see 11.3 of ISO 9075:1992 and 7.2 of this specification).
4. A temporary table, procedure or cursor declared in an abstract module may be an extended temporary table, procedure or cursor (see 8.2 and 8.4 of this specification) only if the keyword **extended** appears in the abstract module declaration.

Ada Semantics

For each abstract module within a compilation unit there is a corresponding Ada library unit package the name of which is the name of the abstract module, that is, AdaID(identifier_1). The Ada construct giving the Ada semantics of any construct declared within an abstract module is declared within the specification of that library unit package. Nothing else appears in the specification of that package.

SQL Semantics

There is an SQL module associated with each abstract module. The SQL construct giving the SQL semantics of any construct declared within an abstract module is declared within that SQL module. The name of the SQL module is implementation defined. The language clause of the SQL module shall specify Ada. If a module authorization clause appears in the abstract module, the identical clause shall appear in the SQL module; otherwise the module authorization clause in the SQL module is implementation defined. See clause 12 of ISO/IEC 9075:1992.

A temporary table declaration is transformed into an SQL temporary table declaration in the same way that a table definition is transformed into an SQL table definition. See 7.2 of this specification.

8.2 Procedures

The procedures discussed in this clause are not associated with a cursor. Cursor procedures are discussed in 8.5 of this specification.

For every procedure declared within an abstract module there is an Ada procedure declared within the library unit package specification corresponding to that abstract module (see 8.1 of this specification) and an

SQL procedure declared within the corresponding SQL module. A call to the Ada procedure results in the execution of the SQL procedure.

```

procedure_declaration ::=
  [extended] procedure identifier_1 [input_parameter_list] is
    statement [status_clause];

statement ::=
  executable_schema_statement
  non_cursor_data_statement
  dynamic_statement
  schema_manipulation_statement
  transaction_statement
  connection_statement
  session_statement
  diagnostic_statement
  extended_statement

```

1. Identifier_1 is the *name* of the procedure.
2. If the statement within a procedure is an insert_statement_row, then the procedure shall not have an input_parameter_list. See 8.3.
3. Schema manipulation, transaction, connection, session, dynamic and diagnostic statements are identical to the identically named statements in ISO/IEC 9075:1992 *except* that any SQL value expressions or search conditions are replaced by the value expressions and search conditions of this specification (see 8.10 and 8.11).
4. A statement may be an extended_statement only if the keyword **extended** appears in the procedure declaration. If the keyword **extended** appears in the procedure declaration, then the keyword **extended** shall appear within the definition of the module in which the procedure is declared.

Ada Semantics

Each procedure declaration P shall be assigned an Ada procedure declaration P_{Ada} in a manner that satisfies the following constraints:

- If P is declared within the declaration of an abstract module M , then P_{Ada} is declared directly within the library unit package specification M .
- The simple name of P_{Ada} is $AdaID(Identifier_1)$, the name of P .

The parameter profile of the Ada procedure is defined as follows:

1. Let the input_parameter_list of the procedure contain k input parameters (for some $k \geq 0$), given by $INP_1, INP_2, \dots, INP_k$. Then the i^{th} parameter declaration in the *Ada_formal_part* of P_{Ada} , denoted $PARM_{Ada}(INP_i)$ for $i \leq k$, is given by

$AdaNAME(INP_i) : in AdaTYPE(INP_i)$

(See 8.6 of this specification.)

2. If the statement within the procedure is a select_statement_single_row, then the $(k+1)^{st}$ parameter in the *Ada_formal_part* of P_{Ada} is a row record. The mode of the row record parameter shall be **in out**.

Let IC be the into_clause appearing (possibly by assumption, see 8.3 of this specification) in the select_statement. Then the name of the row record parameter is $PARM_{Row}(IC)$; the name

of the type of that parameter is $TYPE_{Row}(IC)$. See 8.9 of this specification. If IC contains the keyword **new**, then the declarative region containing the declaration of P_{Ada} shall also contain the declaration of $TYPE_{Row}(IC)$.

The names, types, and order of the components of the row record parameter are determined from the *select_list* within the *select_statement*. Let that list be given by $SP_1; SP_2; \dots; SP_m$. (If the *select_list* takes the form '*' then assume the transformation described in 8.7 of this specification has been applied.) Then the row record type is equivalent in the sense of 3.2.10 and 3.7.2 of ISO/IEC 8652:1995, to a record whose sequence of components is given by the sequence

... $COMP_{Ada}(SP_i)$ [$DBLeng_{Ada}(SP_i)$] ...

where $COMP_{Ada}(SP_i)$ is given by

$AdaNAME(SP_i) : AdaTYPE(SP_i)$

provided that $AdaNAME(SP_i)$ and $AdaTYPE(SP_i)$ are defined (see 8.7 of this specification). The record component $COMP_{Ada}(SP_i)$ is otherwise undefined. The record component $DBLeng_{Ada}(SP_i)$ is given by

$DBLengNAME(SP_i) : Ada_Indicator_Type$

where $Ada_Indicator_Type$ is the type $SQL_Standard_Indicator_Type$ (see 12.3.8.a.iii of ISO/IEC 9075:1992), provided that $DBLengNAME(SP_i)$ is defined; otherwise this component is not present.

Note: $COMP_{Ada}(SP_i)$ is undefined only if the i^{th} select parameter is improperly written; whereas $DBLeng_{Ada}(SP_i)$ is undefined when the i^{th} select parameter does not have a **dblength** phrase. See 8.7 of this specification.

3. If the statement within the procedure is an *insert_statement_row* and it is *not* the case that the *insert_value_list* is present and consists solely of literals and constants, then the first parameter is a row record. The mode of the record parameter is **in**.

Let IC be the *insert_from_clause* appearing (possibly by assumption, see 8.3 of this specification) in the statement. Then the name of the row record parameter is $PARM_{Row}(IC)$; the name of the type of that parameter is $TYPE_{Row}(IC)$. See 8.3 of this specification. If IC contains the keyword **new**, then the declarative region containing the declaration of P_{Ada} also contains the declaration of $TYPE_{Row}(IC)$.

The names, types, and order of the components of the record type are determined from the *insert_column_list* and *insert_value_list*. So, let C_1, C_2, \dots, C_m be the subsequence of *insert_column_specifications* appearing in an *insert_column_list* such that the corresponding element of the *insert_value_list* is not a literal or constant reference. Then the row record type is equivalent in the sense of 3.2.10 and 3.7.2 of ISO/IEC 8652:1995, to the record whose i^{th} component $COMP_{Ada}(C_i)$ for $1 \leq i \leq m$ is given by

$AdaNAME(C_i) : AdaTYPE(C_i)$

(See 8.8 of this specification).

4. If the statement within the procedure is an *extended_statement*, see 6.6 of this specification; for extended parameter lists, see 8.6 of this specification.
5. For all procedures, regardless of statement type, if a *status_clause* appears in the procedure declaration, then the final parameter is a status parameter of mode **out**. For the name and type of this parameter, see 7.1.8 and 8.12 of this specification.

SQL Semantics

Each procedure declaration P shall be assigned an SQL procedure P_{SQL} within the SQL module for the abstract module in which the procedure appears. P_{SQL} has three parts:

1. An *SQL_procedure_name*. This is implementation defined.
2. A list of *SQL_parameter_declarations*. An SQLSTATE parameter is declared for every SQL procedure unless a *status_clause* appears in the procedure declaration that references a status map specifying **sqlcode**, in which case an SQLCODE parameter is declared. (See 7.1.8 and 8.12 of this specification.)
3. If an input parameter list is present, then the SQL parameters derived from the *input_parameter_list* of the procedure, as described in 8.6 of this specification, appear in the parameter declarations of P_{SQL} .
4. Other parameters depend on the type of the statement within the procedure P .
 - a. If the statement is an *insert_statement_row*, then the SQL parameters are determined by the *insert_values* in *insert_value_lists* that are *column_names*. See 8.8 of this specification.
 - b. If the statement is a *select_statement_single_row*, then the SQL parameter declarations for P_{SQL} are determined by the *select_list* of the *select_statement_single_row*, as described in 8.7 of this specification.
 - c. If the statement is an *extended_statement*, see 6.6 of this specification.
5. An *SQL_SQL_procedure_statement* (see 12.5 in ISO/IEC 9075:1992). This is derived from the statement in the procedure declaration by applying the transformations SQL_{VE} to any value expression and SQL_{SC} to any search condition appearing in the statement in the procedure. Certain statement types have specific transformation rules. See 8.3.

Interface Semantics

A call to the Ada procedure P_{Ada} shall have effects that cannot be distinguished from the following:

1. The procedure P_{SQL} is executed in an environment in which the values of parameters $PARM_{SQL}(INP)$ and $INDIC_{SQL}(INP)$ (see 8.6 of this specification) are set from the value of $PARM_{Ada}(INP)$ (see Ada Semantics above) according to the rule for input parameters of 7.3 of this specification, for every input parameter INP in the *input_parameter_list* of the procedure. Similarly, for each *insert_value* C that is a *column_name* in an *insert_value_list* of an *insert_statement_row*, the value of $PARM_{SQL}(C)$ and $INDIC_{SQL}(C)$ are set from the value of $COMP_{Ada}(C)$, according to the rule of 7.3 of this specification. See 8.8 of this specification.
2. Standard post processing, as described in 6.5 of this specification, is performed.
3. If the value of the SQL status parameter indicates successful completion or if it is handled by a status map attached to the procedure (see 6.5 of this specification) and is defined by the SQL-implementation to permit the transmission of data and the statement within the procedure is a *select_statement*, then the value of the component of the row record parameter components $COMP_{Ada}(SP_i)$ and $DBLeng_{Ada}(SP_i)$ are set from the values of the actual parameters associated with the SQL formal parameters $PARM_{SQL}(SP_i)$ and $INDIC_{SQL}(SP_i)$ (see 8.7 of this specification), according to the rule for output parameters of 7.3 of this specification.

Note: Examples

VERSION 2

The following are examples of procedure declarations. The first is a declaration of a procedure with no input parameters.

```
procedure Parts_Suppliers_Commit is  
    commit work;
```

The above declaration produces the following Ada procedure specification in the abstract interface.

```
procedure Parts_Suppliers_Commit;
```

The next procedure declaration contains an input parameter and a status clause.

```
procedure Delete_Parts (  
    Input_Pname named Part_Name : Part_Names) is  
  
    delete from P  
        where PNAME = Input_Pname  
        status Operation_Map named Delete_Status  
    ;
```

The above declaration produces the following Ada procedure specification in the abstract interface.

```
procedure Delete_Parts (  
    Part_Name      : in Part_Names_Type;      -- 8.6: Ada Semantics #1  
                                           -- 8.2: Ada Semantics #1, #2  
    Delete_Status  : out Operation_Status); -- 8.2: Ada Semantics #5
```

A somewhat more complex example, involving a row record: the SAMeDL procedure:

```
procedure Insert_Redparts is  
    insert into P  
        (PNO named Part_Number,  
         PNAME named Part_Name,  
         COLOR,  
         CITY)  
    from Red_Parts  
    values (PNO, PNAME, Red, CITY);
```

produces the following Ada declarations:

```
type Insert_Redparts_Row_Type -- 8.2, Ada semantics #3, 8.9  
is record  
    Part_Number : Part_Numbers_Type;  -- 7.2.1  
    Part_Name   : Part_Names_Type;  
    City        : City_Names_Type;  
end record;  
  
procedure Insert_Redparts  
    (Red_Parts : in Insert_Redparts_Row_Type);
```

The color of all parts inserted using the Insert_Redparts procedure will be red. The weight of all such parts will be null. See the examples in 7.2 of this specification. The number, name and city of those parts are specified at run time.

End Examples

8.3 Non cursor data statements

This clause describes the concrete syntax of data statements that are not cursor-oriented. The text of the SQL statement derived from the text of a statement is defined.

```
non_cursor_data_statement ::=  
    delete_statement_searched
```

VERSION 2

```
insert_statement
insert_statement_row
update_statement_searched
select_statement_single_row

insert_statement_row ::=
  insert into table_name [(insert_column_list)]
  [insert_from_clause] values [(insert_value_list)]
```

Other than the `insert_statement_row`, the non cursor data statements are identical to the similarly named SQL data statements (see 12.5 of ISO/IEC 9075:1992) except that

1. Value expressions as defined in 8.10 replace SQL value expressions;
2. Search conditions as defined in 8.11 replace SQL search conditions;
3. The select list defined in 8.7 and the into clause defined 8.9 of this specification replace the SQL select list and select target list in a single row select statement. However, the select list of any contained subquery is not replaced in this way.

In the following discussion, let *ProcName* be the name of the procedure in which the statement appears.

1. If no `insert_from_clause` appears within an `insert_statement_row`, then the following clause is assumed:

```
from Row : new ProcName_Row_Type
```

If an `insert_from_clause` that does not contain a `record_id` appears in an `insert_statement_row`, the `record_id`

```
: new ProcName_Row_Type
```

is assumed. See 8.9 of this specification.

2. If no `into_clause` appears within a `select_statement_single_row`, then the following clause is assumed:

```
into Row : new ProcName_Row_Type
```

If an `into_clause` which does not contain a `record_id` appears in a `select_statement`, the `record_id`

```
: new ProcName_Row_Type
```

is assumed. See 8.9 of this specification.

3. This rule applies to all forms of insert statements. If an `insert_column_list` is not present, then a column list consisting of all columns defined for the table denoted by *SQL_table_name* is assumed, in the order in which the columns were declared (8.7.3 of ISO/IEC 9075:1992).

Note: Use of the empty `insert_column_list` is considered poor programming practice. The interpretation of the empty `insert_column_list` is subject to change as the database design changes. Programs that use an empty `insert_column_list` present maintenance difficulties not presented by programs supplying an `insert_column_list`.

4. If the statement is an `insert_statement_row`, then
 - a. If the `insert_value_list` is not present, then a list consisting of the sequence of column names in the `insert_column_list` is assumed.
 - b. The `insert_column_list` and `insert_value_list` shall conform, as described in 8.8 of this specification.
5. In an `insert_statement` not an `insert_statement_row`, let C_1, C_2, \dots, C_m be the columns appearing in the `SQL_insert_column_list` (possibly by assumption, see above). The query_expression appearing in the statement is equivalent (see ISO/IEC 9075:1992 7.5 and 7.10) to one of the

form

$E_1 [OP E_2]$

where OP is a set operator and the two query_expressions, E_1 and E_2 , define tables of the same length. Let that length be n and let C^1_i and C^2_i for $1 \leq i \leq n$ be the sequence of columns of those tables. We may assume, by recursion over the number of set operators, that $DOMAIN(C^1_i)$ and $DOMAIN(C^2_i)$ are defined. Then

- a. $m = n$, that is, the lists are of the same length; and
 - b. $DOMAIN(C^1_i) = DOMAIN(C^2_i) = DOMAIN(C_i)$.
6. The following apply to update statements. Let

$C = v$

be a set_item within an update_statement. Let D be $DOMAIN(C)$. Then

- a. If v is **null**, D shall not be a not null only domain.
- b. Otherwise, if v is a value_expression, v shall conform to D (see 6.4 of this specification).

SQL Semantics

This clause describes the text of an SQL statement corresponding to the statement within a procedure.

1. The transformations SQL_{VE} and SQL_{SC} are applied to all value expressions and search conditions in the statement.
2. The insert_statement_row is transformed into an $SQL_insert_statement$ by transforming each insert_value_list and the insert_column_list as described in 8.8 of this specification and dropping any insert_from clause. The remainder of the statement is unchanged.
3. The select_statement is transformed into an $SQL_select_statement$: single row by
 - a. Replacing the select_list with the SQL_select_list described in 8.7 of this specification.
 - b. Inserting the phrase INTO *target list*, where *target list* is as specified in 8.7 of this specification, and removing the into_clause in the statement, if any.

The remainder of the statement is unchanged.

8.4 Cursor Declarations

```

cursor_declaration ::=
  [extended] cursor identifier_1 [insensitive] [scroll]
    [input_parameter_list]
    for
      query
    ;
  [ is cursor_procedures
  end [identifier_2];]
query ::=
  cursor_specification | extended_cursor_specification
  
```

1. Identifier_1 is the *name* of the cursor. If present, identifier_2 shall be equivalent to identifier_1.
2. No two procedures within a cursor_declaration shall have the same name.
3. For each column C , $INTER_TYPE(DOMAIN(C))$ shall not be **none**. See the Ada and SQL Semantics sections, below.
4. A query may be an extended_cursor_specification only if the keyword **extended** appears in the

VERSION 2

cursor declaration. If the keyword **extended** appears in the cursor declaration, then the keyword **extended** shall appear in the declaration of the module in which the cursor is declared (see 6.6 of this specification).

A cursor_specification is identical to an SQL cursor_specification except that

1. Value expressions as defined in 8.10 replace SQL value expressions;
2. Search conditions as defined in 8.11 replace SQL search conditions;
3. The select list defined in 8.7 of this specification replace the SQL select list in the contained query specification. However, any select list within a subquery of the contained query specification is not replaced in this way. See 7.11 of ISO 9075:1992.

Ada Semantics

If a cursor named *C* is declared within an abstract module named *M*, then a subpackage named *C* shall exist within the Ada package *M* (see 8.1 of this specification). That subpackage shall contain the declarations of the procedures declared in the sequence cursor_procedures. (Some of those procedures may appear by assumption. See 8.5 of this specification.) The text of the procedure declarations is described in 8.5 of this specification.

If the query in a cursor declaration is an extended_cursor_specification, see 6.6 of this specification.

As a consequence of the rules of SQL (see ISO/IEC 9075:1992 7.5 and 7.10), the query_expression within a cursor_specification has a definition in the form

$$E_1 \text{ } [OP \text{ } E_2]$$

where *OP* is a set operator and the two query_expressions, *E*₁ and *E*₂, define tables of the same length. Let that length be *m* and let *C*¹_{*i*} and *C*²_{*i*} for 1 ≤ *i* ≤ *m* be the sequence of columns of those tables. Define the values AdaTYPE_{*i*}, AdaNAME_{*i*}, DOMAIN_{*i*} and DBLNgNAME_{*i*} for the *i*th column of the result, as follows. (We may assume, by recursion over the number of set operators in the query_expression, that these mappings are defined for *C*¹_{*i*} and *C*²_{*i*}. See 8.7 of this specification.)

1. The two columns shall have the same Ada type, that is, AdaTYPE(*C*¹_{*i*}) = AdaTYPE(*C*²_{*i*}) (see 8.7 of this specification). The Ada type of the *i*th parameter, AdaTYPE_{*i*}, is that type. This implies DOMAIN(*C*¹_{*i*}) = DOMAIN(*C*²_{*i*}) and DOMAIN_{*i*} is that domain.

Note: This further implies that either (i) DOMAIN_{*i*} is a not null only domain, or (ii) **not null** is specified either for both or for neither of *C*¹_{*i*} and *C*²_{*i*}.

2. If AdaNAME(*C*¹_{*i*}) = AdaNAME(*C*²_{*i*}), then AdaNAME_{*i*} shall be that name; AdaNAME_{*i*} shall be said to be *specified* if either of *C*¹_{*i*} or *C*²_{*i*} is a select parameter containing a named_phrase or either AdaNAME(*C*¹_{*i*}) or AdaNAME(*C*²_{*i*}) is specified. If AdaNAME(*C*¹_{*i*}) ≠ AdaNAME(*C*²_{*i*}), then if exactly one of AdaNAME(*C*¹_{*i*}) or AdaNAME(*C*²_{*i*}) is specified, AdaNAME_{*i*} shall be that name. Otherwise, AdaNAME_{*i*} = NO_NAME.
3. If either DBLNgNAME(*C*¹_{*i*}) or DBLNgNAME(*C*²_{*i*}) is defined, then both shall be defined and shall be equivalent; DBLNgNAME_{*i*} shall be that name. If neither is defined, then DBLNgNAME_{*i*} is said to be null; otherwise, DBLNgNAME_{*i*} is undefined.

VERSION 2

The type of the row record parameter is equivalent in the sense of 3.2.10 and 3.7.2 of ISO/IEC 8652:1995, to a record type whose sequence of components is given by the sequence

... COMP_{Ada}(C_i) [DBLeng_{Ada}(C_i)] ...

where COMP_{Ada}(SP_i) is given by

AdaNAME_i : AdaTYPE_i

provided that AdaNAME_i and AdaTYPE_i are defined; the record component COMP_{Ada}(C_i) is otherwise undefined. The record component DBLeng_{Ada}(C_i) is given by

DBLNgNAME_i : Ada_Indicator_Type

where Ada_Indicator_Type is the type SQL_Standard.Indicator_Type (see 12.3.8.a.iii in ISO/IEC 9075:1992), *provided* that DBLNgNAME_i is defined. If DBLNgNAME_i is null, this component is not present in the row record parameter.

SQL Semantics

From a cursor_specification, two SQL fragments are derived:

1. A list of *SQL_parameter_declarations*.
2. An *SQL_fetch_target_list*.

These are defined on the basis of the components of the row record parameter as described above.

- There are two SQL parameters declared for each component COMP_{Ada}(C_i). They are PARM_{SQL}(C_i) and INDIC_{SQL}(C_i), where the *SQL_parameter_declaration* declaring PARM_{SQL}(C_i) is

:SQL_{NAME}(C_i) INTER_TYPE(DOMAIN_i)

and the *SQL_parameter_declaration* declaring INDIC_{SQL}(C_i) is

:INDIC_{NAME}(C_i) indicator_type

where SQL_{NAME}(C_i) and INDIC_{NAME}(C_i) are *SQL_identifiers* not appearing elsewhere.

- The target list generated from a select_list is a comma-separated list of *SQL_target_specifications* (6.2 of ISO/IEC 9075:1992). The *i*th *SQL_target_specification* in the target list is

:SQL_{NAME}(C_i) INDICATOR :INDIC_{NAME}(C_i)

Note: All derived target specifications contain indicator parameters, irrespective of the presence or absence of a not_null phrase in the select parameter declaration.

A cursor declaration is transformed into an SQL cursor declaration as follows.

1. The string "declare" is prepended to the cursor declaration and, if present, the keyword **extended** is dropped.
2. The input_parameter_list and cursor_procedures are discarded, as is the keyword **cursor** and the **is ... end** bracket. The cursor name identifier₁ is transformed into SQL_{NAME}(identifier₁).
3. The string "cursor" is inserted immediately after the transformed cursor name, but before the keyword **for**.
4. The select_list of any contained query expression is transformed into an *SQL_select_list* as described in 8.7 of this specification.

VERSION 2

5. The search conditions are transformed using the transform SQL_{SC} of 8.11 of this specification. The remainder of the declaration is unchanged.

Note: Examples

The following note consists of two examples of cursor_declarations: the first contains a simple cursor_declaration, while the second contains a more complex declaration that exercises many of the features of the syntax. In both cases the generated Ada code is shown, annotated with references to the appropriate clauses of the language definition.

The example below is a simple cursor_declaration.

```
cursor Select_Suppliers (Time_Limit :DB_Date_String)
for
  select SNO, SNAME, "STATUS", CITY dblelength -- 5.3
  from S, SP
  where S.SNO = SP.SNO and
  Date_Order = CAST(TIMR_LIMIT As DB_Date);
;
```

This declaration produces the following Ada code.

```
package Select_Suppliers is -- 8.4: Ada Semantics
type Row_Type is record-- 8.5, #5 and #8
  Sno      : Supplier_Numbers_Type;-- 8.5, #3 and #8
  Sname    : Supplier_Names_Type;
  STATUS   : Status_Values_Type;
  City     : City_Names_Type;
  City_Dblelength : SQL_Standard.Indicator_Type;
end record;

procedure Open (Time_Limit : in DB_Date_Strings_Type);-- 8.5: #3

procedure Fetch (-- 8.5: #5
  Row      : in out Row_Type; -- 8.5: #8 and Ada Semantics #3
  Is_Found : out boolean);-- 8.5: #5 and Ada Semantics #6

procedure Close;-- 8.5: #4

end Select_Suppliers;
```

The following is an example of a more complex cursor_declaration.

```
cursor Supplier_Operations (
  Input_City named Supplier_City      : City_Names not null;
  Adjustment named Status_Adjustment : Status_Values not null)
for
  select SNO named Supplier_Number,
         SNAME named Supplier_Name,
         "STATUS" + Adjustment named Adjusted_Status,-- 5.3
         CITY named Supplier_City
  from S
  where CITY = Input_City
;
is
  procedure Open_Supplier_Operations is
    open Supplier_Operations;
  procedure Fetch_Supplier_Tuple is
    fetch Supplier_Operations
    into Supplier_Row_Record : new Supplier_Row_Record_Type
    status My_Map named Fetch_Status;
  procedure Close_Supplier_Operations is
    close; -- optional 'cursor name' omitted
  procedure Update_Supplier_Status (
    Input_Status named Updated_Status : Status_Values not null;
    Input_Adjustment named Adjustment : Status_Values) is

    update S
    set "STATUS" = Input_Status + Input_Adjustment -- 5.3
    where current of Supplier_Operations;
  procedure Delete_Supplier is
    delete from S; -- optional "where current of 'cursor name'" omitted
  end Supplier_Operations;
```

VERSION 2

This declaration produces the following Ada code.

```
package Supplier_Operations is -- 8.4: Ada Semantics
  type Supplier_Row_Record_Type is
    record -- 8.5: Ada Semantics #5 #8
      Supplier_Number : Supplier_Numbers_Type;
    -- 8.5: Ada Semantics #3 #8
      Supplier_Name : Supplier_Names_Type;
      Adjusted_Status : Status_Values_Type;
      Supplier_City : City_Names_Type;
    end record;

  procedure Open_Supplier_Operations (
    Supplier_City : in City_Names_Not_Null; -- 8.5: Ada Semantics
    Status_Adjustment : in Status_Values_Not_Null); -- #1, #3, Modes
  procedure Fetch_Supplier_Tuple (
    Supplier_Row_Record : in out Supplier_Row_Record_Type; -- 8.5
    Fetch_Status : out Operation_Status); -- 8.5
  procedure Close_Supplier_Operations;

  procedure Update_Supplier_Status (
    Updated_Status : in Status_Values_Not_Null; -- 8.5: Ada Semantics #2
    Adjustment : in Status_Values_Type); -- 8.5: Ada Semantics

  procedure Delete_Supplier;
end Supplier_Operations;
```

End Examples

8.5 Cursor Procedures

```
cursor_procedures ::=
  cursor_procedure {cursor_procedure}

cursor_procedure ::=
  [extended] procedure identifier_1
  [input_parameter_list] is
  cursor_statement
  [status_clause]
  ;

cursor_statement ::=
  open_statement | fetch_statement | close_statement |
  update_statement_positioned | delete_statement_positioned |
  extended_cursor_statement
```

The cursor statements are identical to the SQL statements except that

1. Cursor names are optional, as is the “where current of” phrase in the update and delete positioned statements. If the cursor name is present, it shall be equal to the name of the cursor within which the procedure declaration appears.
2. The into keyword and fetch target list of an SQL fetch statement is replaced by an into clause, see 8.9 of this specification.

Furthermore

3. Identifier_1 is the *name* of the procedure.
4. If no open_statement appears in a list of cursor_procedures, the declaration **procedure** “open” **is open**; is assumed.
5. If no close_statement appears in a list of cursor_procedures, the declaration **procedure** “close” **is close**; is assumed.
6. If no fetch_statement appears in a list of cursor_procedures, the declaration **procedure** “fetch” **is fetch status standard_map**; is assumed. See 7.1.8 of this specification.

VERSION 2

7. The restrictions that apply to the `set_items` of a non-cursor `update_statement` (see 8.3 of this specification) also apply to the `set_items` of a `cursor_update_statement`.
8. If no `into_clause` appears within a `fetch_statement`, then the following clause is assumed:

`into Row : new Row_Type`

If an `into_clause` which does not contain a `record_id` appears in a `fetch_statement`, the `record_id`

`: new Row_Type`

is assumed. See 8.9 of this specification.

9. A `cursor_statement` may be an `extended_cursor_statement` only if the keyword **extended** appears in the `cursor_procedure` declaration. If the keyword **extended** appears in the `cursor_procedure` declaration, then the keyword **extended** shall appear within the declaration of the cursor in which the `cursor_procedure` is declared.

Ada Semantics

Each procedure declaration P that appears in or is assumed to appear in a `cursor_procedures` list shall be assigned an Ada procedure declaration P_{Ada} that satisfies the following constraints.

- If P is declared within the declaration of a cursor named C , then P_{Ada} shall be declared within the specification of an Ada subpackage named C .
- The simple name of P_{Ada} is `AdaID(identifier_1)`, the name of P .

Note: The default open, fetch and close procedures use delimited identifiers (see 5.3 of this specification) as their procedure names.

The parameter profiles of the Ada procedures depend in part on the statement within the procedure, as follows:

1. For `open_statements`: Let $INP_1, INP_2, \dots, INP_k$ $k \geq 0$ be the list of input parameters in the `input_parameter_list` of the `cursor_declaration` within which the procedure appears. For all other statements, let this list be list of input parameters in the `input_parameter_list` of the procedure in which the statement appears. Then $PARM_{Ada}(INP_i)$, the i^{th} parameter of the `Ada_formal_` part, is of the form

`AdaNAME(INPi) : in AdaTYPE(INPi)`

for

$1 \leq i \leq k$

(see 8.6 of this specification).

2. For `fetch_statements`: The first parameter after any resulting from the input parameter list is a row record parameter of mode **in out**. The names, order, and types of the components of the type of this parameter are described in 8.2 and 8.4 of this specification. Let IC be the `into_`-clause appearing (possibly by assumption) in the `fetch_statement`. Then the name of the row record formal parameter is $PARM_{Row}(IC)$, and the type of that parameter is $TYPE_{Row}(IC)$. See 8.9 of this specification. If IC contains the keyword **new**, then the declarative region containing the declaration of P_{Ada} shall contain the declaration of $TYPE_{Row}(IC)$.
3. For `extended_cursor_statements`, see 6.6 of this specification. For extended parameter lists, see 8.6 of this specification.

4. For all statement types: If a `status_clause` referencing a status map that contains a **uses** appears in the procedure declaration, then the final parameter is a status parameter of mode **out**. For the name and type of this parameter, see 7.1.8 and 8.12 of this specification.

SQL_Semantics

Each procedure P that appears in or is assumed to appear in a `cursor_procedures` list shall be assigned an SQL procedure P_{SQL} within the SQL module for the abstract module within which the `cursor_procedures` list appears. P_{SQL} has three parts:

1. An `SQL_procedure_name`. This is implementation defined.
2. A list of `SQL_parameter_declarations`. An `SQLSTATE` parameter is declared for every SQL procedure unless a status clause appears in the procedure declaration that references a status map specifying **sqlcode**, in which case an `SQLCODE` parameter is declared. (See 7.1.8 and 8.12 of this specification.) Other parameters depend on the type of the statement within the procedure P .
 - a. If the statement is an `open_statement`, then the SQL parameters derived from the `input_parameter_list` of the `cursor_declaration`, otherwise the SQL parameters derived from the `input_parameter_list` of the procedure, if any, as described in 8.6 of this specification appear in the parameter declarations of P_{SQL} .
 - b. If the statement is a `fetch_statement`, then the SQL parameters determined by the query of the `cursor_declaration` (as described in 8.4) of this specification appear in the parameter declarations of P_{SQL} .

The order of the parameters within the list is implementation defined.

3. An `SQL_SQL_data_statement` or `SQL_SQL_data_change_statement`. (See 12. in ISO/IEC 9075:1992.) This is derived from the statement in the procedure declaration, as follows.
 - a. If the statement is an `open_statement`, then the `SQL_open_statement` is **open** `SQL_NAME(C)`, where C is the cursor name.
 - b. If the statement is a `close_statement`, then the `SQL_close_statement` is **close** `SQL_NAME(C)`, where C is the cursor name.
 - c. If the statement is the `delete_statement_positioned`

```
delete from Id_1 [where current of C]
```

then the `SQL_delete_statement_positioned` is identical, up to the addition of the where phrase, **where current of** `SQL_NAME(C)`, replacing the where phrase of the `cursor_delete_statement`, if present.
 - d. If the statement is the `update_statement_positioned`

```
update Id_1
set set_items
[where current of C]
```

then the `SQL_update_statement_positioned` is formed by applying the transformation `SQLVE` defined in 8.10 of this specification to the value expressions in the `set_items` of the statement and appending or replacing the where phrase so as to read **where current of** `SQL_NAME(C)`.
 - e. If the statement is a `fetch_statement`, then the `SQL_fetch_statement` is

```
fetch SQL_NAME(C) into target list
```

where C is the cursor name and *target list* as described in 8.4 of this specification.

- f. If the statement is an `extended_statement`, see 6.6 of this specification.

Interface Semantics

A call to the Ada procedure P_{Ada} shall have effects that can not be distinguished from the following:

1. The procedure P_{SQL} is executed in an environment in which the values of parameters $PARM_{SQL}(INP)$ and $INDIC_{SQL}(INP)$ (see 8.6 of this specification) are set from the value of $PARM_{Ada}(INP)$ (see Ada semantics above) according to the rule of 7.3 of this specification for every input parameter, INP , in either the `input_parameter_list` of the `cursor_declaration` for open procedures, or the `input_parameter_list` of the procedure itself for update procedures.
2. Standard post processing, as described in 6.5 of this specification, is performed.
3. If the value of the SQL status parameter indicates successful completion or if it is handled by a status map attached to the procedure (see 6.5 of this specification) and is defined by the SQL-implementation to permit the transmission of data and the statement within the procedure is a `fetch_statement`, then the value of the row record components $COMP_{Ada}(SP_i)$ and $DBLeng_{Ada}(SP_i)$, are set from the values of the actual parameters associated with the SQL formal parameters $PARM_{SQL}(SP_i)$ and $INDIC_{SQL}(SP_i)$ (see 8.4 and 8.7 of this specification) according to the rule of 7.3 of this specification.

8.6 Input Parameter Lists

Input parameter lists declare the parameters of the procedure, cursor, or cursor procedure declaration in which they appear. The list consists of parameter declarations that are separated with semicolons, in the manner of Ada formal parameter declarations.

Each parameter declaration of a procedure or cursor procedure P is represented as an *Ada_parameter_specification* within the *Ada_formal_part* of the procedure P_{Ada} ; each parameter declaration within a cursor declaration is represented as an *Ada_parameter_specification* within the *Ada_formal_part* of the Ada open procedure. The parameter is also represented as either one or two *SQL_parameter_declarations* within the *SQL_procedure* P_{SQL} . The second SQL parameter declaration, if present, declares the indicator variable for the parameter (see 4.17.2 of ISO/IEC 9075:1992).

The order of parameter specification within the *Ada_formal_part* is given by the order within the `input_parameter_list`. The order of the *SQL_parameter_declarations* within the list of declarations in the SQL procedure is implementation defined.

```
input_parameter_list ::=
    (parameter {; parameter})

parameter ::=
    parameter_name [named_phrase] :
        [in] [out] domain_reference [not null]

parameter_name ::= identifier
```

Ada Semantics

Let INP be a parameter the textual representation of which is given by

VERSION 2

Id_1 [**named** Id_2] : [**in**] [**out**] [$Id_3.$] Id_4 [**not null**]

then Id_1 is the *name* of the parameter.

The domain associated with INP, denoted DOMAIN(INP), is the domain referenced by [$Id_3.$] Id_4 . Let DOMAIN(INP) = D ; then DATACLASS(INP) = DATACLASS(D). INTER_TYPE(D) shall not be **none**.

The functions AdaNAME and AdaTYPE are defined on parameters as follows:

1. If Id_2 is present in the definition of INP, then AdaNAME(INP) = AdaID(Id_2) otherwise, AdaNAME(INP) = AdaID(Id_1). For no two parameters, INP_1 and INP_2 , in an input parameter list shall AdaNAME(INP_1) = AdaNAME(INP_2).
2. AdaTYPE(INP) shall be the name of a type within the domain identified by the domain_reference [$Id_3.$] Id_4 . If **not null** appears within the textual representation of INP, or the domain identified by the domain_reference is not null only, then AdaTYPE(INP) shall be the name of the not null-bearing type within the identified domain; otherwise it shall be the name of the null-bearing type within that domain (see 7.1.3 of this specification).

The optional **out** may occur only in a parameter that is associated with a procedure or cursor that is extended. The optional **in**, however, may be included in any parameter declaration.

Given INP as defined above, define mode(INP) to be

- *in*, if INP either contains (1) the optional **in**, but not the optional **out**, or (2) neither **in** nor **out**.
- *out*, if INP contains **out** but not **in**.
- *in out*, if INP contains both **in** and **out**.

Then the generated parameter, PARM_{Ada}(INP), in the Ada_formal_part is of the form

AdaNAME (INP) : mode (INP) AdaTYPE (INP) ;

SQL Semantics

Let INP be as given above and let D be the domain referenced by [$Id_3.$] Id_4 . The SQL parameter PARM_{SQL}(INP) is declared by the following SQL_parameter_declaration

:SQL_{NAME}(Id_1) INTER_TYPE(D)

where INTER_TYPE(D) is given in 7.1.3 of this specification. If **not null** does *not* appear within the textual representation of INP, and [$Id_3.$] Id_4 does *not* identify a not null only domain, then the parameter INDIC_{SQL}(INP) is defined and is declared by the SQL_parameter_declaration

:INDIC_{NAME}(INP) *indicator_type*

where *indicator_type* is the implementation-defined type of indicator parameters (Syntax Rule 1 of 6.2 of ISO/IEC 9075:1992). The name INDIC_{NAME}(INP) shall not appear as the name of any other parameter of the enclosing procedure.

8.7 Select Parameter Lists

Select parameter lists serve to inform the DBMS of what data are to be retrieved by a select or fetch statement. They also specify the names and types of the components of a record type—the so called row record type—which appears as the type of a formal parameter of Ada procedure declarations for select and fetch statements.

```
select_list ::= * | select_parameter {, select_parameter}
select_parameter ::=
  value_expression [named_phrase] [not null]
  [dblength [named_phrase]]
```

1. The select list star (“*”) is equivalent to a sequence of select parameters described as follows:
Let T_1, T_2, \dots, T_k be the list of the exposed table names in the table expression from clause for the query specification in which the select list appears (see 7.9 of ISO/IEC 9075:1992). Let U_i , for $1 \leq i \leq k$ be defined as $S_i _ V_i$ if T_i is of the form $S_i.V_i$ (i.e., S_i is a schema_module_name, and V_i is a table name); otherwise, U_i is T_i . In other words, U_i is T_i with every dot “.” replaced by an underscore “_”. Let $A_{i,1}, A_{i,2}, \dots, A_{i,m_i}$ be the names of the columns of the table named T_i . Then the select list is given by the sequence $T_1.A_{1,1}$ **named** $U_1 _ A_{1,1}$, $T_1.A_{1,2}$ **named** $U_1 _ A_{1,2}$, \dots , $T_i.A_{i,j}$ **named** $U_i _ A_{i,j}$, \dots , $T_k.A_{k,m}$ **named** $U_k _ A_{k,m}$. That is, the columns are listed in the order in which they were defined (see 7.2 of this specification) within the order in which the tables were named in the from clause.

Notes: This definition differs from that given in syntax rule 3.b of 7.9 in ISO/IEC 9075:1992 in specifying that the column references are qualified by table name or correlation name. The table that is described in the cited rule of SQL has anonymous columns. The record type being described must have well-defined component names.

Use of * as a select list in an abstract module is considered poor programming practice. The interpretation of * is subject to change with time as the database design changes. Programs that use * as a select list present maintenance difficulties not presented by programs supplying a proper select list.

In the following discussion, assume that a select list ‘*’ has been replaced by its equivalent list, as described above.

2. If the keyword **dblength** is present, then value_expression shall have the dataclass **character**.
3. Let VE be the value_expression appearing in a select_parameter. DOMAIN(VE) shall not be NO_DOMAIN and VE shall conform to DOMAIN(VE).

Ada Semantics

Let SP be a select parameter written as

```
VE [named Id_1] [not null]
[dblength [named Id_2] ]
```

SP is assigned the Ada type name AdaTYPE(SP), the Ada name AdaNAME(SP) and the dblength name DBLnGNAME(SP) as follows:

- Let $\text{DOMAIN}(VE) = D$ (see 8.10 of this specification) where $D \neq \text{NO_DOMAIN}$. If **not null** appears in SP, or D is a not null only domain, then AdaTYPE(SP) is the name of the not null-bearing type name within the domain D ; else AdaTYPE(SP) is the name of the null-bearing type within the domain D .

VERSION 2

- If $\text{DOMAIN}(\text{VE}) = \text{NO_DOMAIN}$ then $\text{AdaTYPE}(\text{SP})$ is undefined.
 - If Id_1 appears in SP, then $\text{AdaNAME}(\text{SP}) = \text{AdaID}(\text{Id}_1)$; else $\text{AdaNAME}(\text{SP}) = \text{AdaNAME}(\text{VE})$ (see 8.10 of this specification).
 - If the **dblength** phrase appears in SP, then
 - if Id_2 is present then $\text{DBLNgNAME}(\text{SP}) = \text{AdaID}(\text{Id}_2)$
 - else, $\text{DBLNgNAME}(\text{SP}) = \text{AdaNAME}(\text{SP})_DbLength$
- Otherwise, $\text{DBLNgNAME}(\text{SP})$ is undefined.
- $\text{DBLNgNAME}(\text{SP})$ and $\text{AdaNAME}(\text{SP})$ shall not appear as either $\text{DBLNgNAME}(\text{SP}_i)$ or as $\text{AdaNAME}(\text{SP}_i)$ for any other select_parameter SP_i within the select_list that contains SP.

SQL Semantics

From a select_list, three SQL fragments are derived:

1. An *SQL_select_list* within a select statement or cursor declaration.
2. A list of *SQL_parameter_declarations*.
3. Either an *SQL_select_target_list* within a select statement or an *SQL_fetch_target_list* within a fetch statement.

An *SQL_select_list* is derived from a select_list as follows:

- The select_list * becomes the *SQL_select_list* *.
- Otherwise, suppose $\text{SP}_1, \text{SP}_2, \dots, \text{SP}_n$ is a select_list, where SP_i is given by:

$$\text{VE}_i \text{ [named Id}_1\text{]}_i \text{ [not null]}_i \text{ [dblength}_i \text{ [named Id}_2\text{]}_i \text{]}$$

The *SQL_select_list*, $\text{SP}'_1, \text{SP}'_2, \dots, \text{SP}'_n$ is formed by setting SP'_i to $\text{SQL}_{\text{VE}}(\text{VE}_i)$.

For the purpose of defining the *SQL_parameter_declarations* and target list generated from a select_list, let $\text{SP}_1, \text{SP}_2, \dots, \text{SP}_n$ be the select_list supplied or the select_list that replaced the select_list * as described above. Let each SP_i be as given above. Then

- There are two SQL parameters associated with each select_parameter, SP_i . They are $\text{PARM}_{\text{SQL}}(\text{SP}_i)$ and $\text{INDIC}_{\text{SQL}}(\text{SP}_i)$, where the *SQL_parameter_declaration* declaring $\text{PARM}_{\text{SQL}}(\text{SP}_i)$ is

$$:\text{SQL}_{\text{NAME}}(\text{SP}_i) \text{ INTER_TYPE}(\text{DOMAIN}(\text{VE}_i))$$

and the *SQL_parameter_declaration* declaring $\text{INDIC}_{\text{SQL}}(\text{SP}_i)$ is

$$:\text{INDIC}_{\text{NAME}}(\text{SP}_i) \text{ indicator_type}$$

where $\text{SQL}_{\text{NAME}}(\text{SP}_i)$ and $\text{INDIC}_{\text{NAME}}(\text{SP}_i)$ are *SQL_identifiers* not appearing elsewhere.

- The target list generated from a select_list is a comma-separated list of *SQL_target_specifications* (6.2 of ISO/IEC 9075:1992). The i^{th} *SQL_target_specification* in the target list is

$$:\text{SQL}_{\text{NAME}}(\text{SP}_i) \text{ INDICATOR } :\text{INDIC}_{\text{NAME}}(\text{SP}_i)$$

Note: All derived target specifications contain indicator parameters, irrespective of the presence or absence of a not_null phrase in the select parameter declaration.

8.8 Insert Column Lists and Insert Value Lists

```

insert_column_list ::=
  insert_column_specification {, insert_column_specification}

insert_column_specification ::=
  column_name [named_phrase] [not null]

insert_value_list ::=
  insert_value {, insert_value}

insert_value ::=
  null | constant_reference
  literal | column_name
  domain_parameter_reference

```

Each column_name within an insert_column_list shall specify the name of a column within the table into which insertions are to be made by the enclosing insert_statement_row. (See 8.3 of this specification and 13.8 in ISO/IEC 9075:1992.)

Let C be the insert_column_specification

```
Col [named Id] [not null]
```

Then AdaNAME(C) is defined to be AdaID(Id), if Id is present; otherwise it is AdaID(Col). Let DOMAIN(C) = DOMAIN(Col) = D be the domain assigned to the column named Col . If **not null** appears in C , or D is a not null only domain, then AdaTYPE(C) is the name of the not null-bearing type within the domain D ; otherwise, AdaTYPE(C) is the null-bearing type within the domain D .

INTER_TYPE(D) shall not be **none**.

Let $CL = C_1, \dots, C_m$ be an insert_column_list; let $IL = V_1, \dots, V_n$ be an insert_value_list. CL and IL are said to conform if:

1. $m = n$, that is, the length of the two lists is the same.
2. For each $1 \leq i \leq m$, if V_i is
 - a. **null**, then DOMAIN(C_i) shall not be a not null only domain.
 - b. A literal or a reference to either a constant or a domain parameter, then V_i shall conform to DOMAIN(C_i) (see 6.4 of this specification).
 - c. A column_name, then V_i shall be identical to the column_name in C_i .

Ada Semantics

The insert_value_list of an insert_statement_row defines the components of an Ada record type declaration. The names, types and order of those components are defined in 8.2 of this specification on the basis of the functions AdaNAME and AdaTYPE described above. For the name of the record type and its place of declaration, see 8.9 of this specification.

Note: If the insert_value_list contains only constants and literals, then the Ada procedure corresponding to the procedure containing the insert_statement_row statement of which these lists form a part does not have a row record parameter. See 8.2 of this specification.

VERSION 2

SQL Semantics

There is an SQL parameter corresponding to each insert_value in an insert_value_list that is a column_name. So let C be such a column_name. Then the SQL parameter declaration $\text{PARM}_{\text{SQL}}(C)$ is given by

$:\text{SQL}_{\text{NAME}}(C) \text{ INTER_TYPE}(\text{DOMAIN}(C))$

where

1. $\text{SQL}_{\text{NAME}}(C)$ is an implementation-defined identifier that does not otherwise appear in the enclosing procedure.
2. $\text{INTER_TYPE}(\text{DOMAIN}(C))$ is as defined in 7.1.3 of this specification.

Assuming the conformance of the insert_value_list with the insert_column_list of the enclosing statement (see above), an entry of the form

$C \text{ [named } Id] \text{ [not null]}$

appears in the insert_column_list. If **not null** does *not* appear *and* the domain $\text{DOMAIN}(C)$ is *not* not null only, then the parameter $\text{INDIC}_{\text{SQL}}(C)$ is defined and the parameter declaration

$:\text{INDIC}_{\text{NAME}}(C) \text{ indicator_type}$

also appears in the list of SQL parameter declarations, where

1. $\text{INDIC}_{\text{NAME}}(C)$ is an implementation-defined identifier that does not otherwise appear in the enclosing procedure.
2. *indicator_type* is the implementation-defined type of indicator parameters (6.2 in ISO/IEC 9075:1992).

An insert_column_list and insert_value_list pair are transformed into an $\text{SQL_insert_column_list}$ and $\text{SQL_table_value_constructor}$ pair (see 7.2 and 13.8 of ISO/IEC 9075:1992) as follows:

1. An insert_column_list is transformed into an $\text{SQL_insert_column_list}$ by the removal of all *named_phrases* and *not_null* phrases that appear in it.
Note: This implies that the empty insert_column_list is transformed into the empty $\text{SQL_insert_column_list}$.
2. Each insert_value_list is transformed into an $\text{SQL_table_value_constructor}$ by replacing each list element as follows:
 - a. A literal, or **null** (but excluding any enumeration literal), is replaced by itself; i.e., is unchanged.
 - b. A constant_reference or enumeration literal k is replaced by a textual representation of its database value $\text{SQL}_{\text{VE}}(k)$ (see 7.3 and 8.10 of this specification).
 - c. A column_name Col_i is replaced by

$:\text{SQL}_{\text{NAME}}(Col_i) \text{ [INDICATOR :INDIC}_{\text{NAME}}(C_i)]$

where the INDICATOR phrase appears whenever the indicator parameter, $\text{INDIC}_{\text{SQL}}(C_i)$, is defined (see above).

The resulting list is enclosed in parentheses.

8.9 Into_Clause and Insert_From_Clause

An `into_clause` is used within a `select_statement` or a `fetch_statement`, and an `insert_from_clause` is used within an `insert_statement_row` to explicitly name the row record parameter of those statements and/or the type of that parameter.

```

into_clause ::=
  into into_from_body
insert_from_clause ::=
  from into_from_body
into_from_body ::=
  identifier_1 : record_id |
  identifier_1 |
  : record_id
record_id ::=
  new identifier_2 | record_reference

```

Ada Semantics

Define the string $\text{PARM}_{\text{Row}}(IC)$ as follows, where IC is an `into_clause` or `insert_from_clause`.

1. If `identifier_1` appears in IC , then $\text{PARM}_{\text{Row}}(IC) = \text{AdaID}(\text{identifier}_1)$.
2. Otherwise, if `record_id` is a `record_reference` referencing the record declaration R , then $\text{PARM}_{\text{Row}}(IC) = \text{AdaNAME}(R)$. See 7.1.5 of this specification.
3. Otherwise, $\text{PARM}_{\text{Row}}(IC) = \text{Row}$.

Define $\text{TYPE}_{\text{Row}}(IC)$ as follows:

1. If `record_id` has the form


```

new identifier_2

```

 then $\text{TYPE}_{\text{Row}}(IC) = \text{AdaID}(\text{identifier}_2)$.
2. Otherwise, $\text{TYPE}_{\text{Row}}(IC)$ is the record type referenced by the `record_reference`.

Note: The assumptions made about `into_clause` and `insert_from_clause` in 8.3 and 8.5 of this specification are sufficient to ensure that every such clause contains a `record_id`, possibly by assumption. Therefore, the case of a missing `record_id` need not be considered in the definition of $\text{TYPE}_{\text{Row}}(IC)$. If the `record_id` is a `record_reference`, then the names, types, and order of the components of the referenced record must exactly match the names, types, and order of the components of the record type declaration that would have been generated had the `record_id` been “new identifier” (see 8.4, 8.2, and 8.7 of this specification).

Note: Examples

The following is a set of example procedure declarations that illustrate various uses of `into` and `from` clauses. It is assumed that each of these procedures is declared within an abstract module, and that any enumeration, record, and status map declarations used are visible at the point at which each procedure is declared. Declarations for these constructs can be found in 7.1.6, 7.1.5, and 7.1.8 of this specification, respectively. In addition, it is assumed that the abstract modules in which these procedures are declared have direct visibility to the contents of the *Parts_Suppliers_Database* schema module shown in the examples in 7.2 of this specification.

VERSION 2

The two examples below illustrate the use of a previously declared record object in the **into** clauses of select statements. The cursor and the select statement retrieve the same object, namely a part. The row record, **Parts_Row_Record_Type** (see 7.1.5 of this specification), contains the definition of the part abstraction.

```
cursor Parts_By_City (
  Input_City named Part_Location : City not null)

  for
    select PNO named Part_Number not null,
           PNAME named Part_Name,
           COLOR,
           Weight_In_Grams(WEIGHT * Grams_In_Pound) named Weight,
           -- 8.10, 7.1.4
           CITY
    into Parts_By_City_Row : Parts_Row_Record_Type
    from P
    where CITY = Input_City
  ;

procedure Parts_By_Number (
  Input_Pno named Part_Number : Pno not null) is
  select PNO named Part_Number not null,
        PNAME named Part_Name,
        COLOR,
        Weight_In_Grams(WEIGHT * Grams_In_Pound) named Weight,
        -- 8.10, 7.1.4
        CITY
  into Parts_By_Number_Row : Parts_Row_Record_Type
  from P
  where PNO = Input_Pno
  status Operation_Map named Parts_By_Number_Status
  ;
```

The above declarations produce the following Ada declarations at the abstract interface.

```
package Parts_By_City is

  procedure Open (
    Part_Location : in City_Names_Not_Null);-- 8.5: Ada Semantics

  procedure Fetch (
    Parts_By_City_Row : in out Parts_Row_Record_Type;
    -- 8.5: Ada Semantics #3
    Is_Found          : out boolean);-- 8.5: Ada Semantics #6

  procedure Close;
end Parts_By_City;

  procedure Parts_By_Number (
    Part_Number      : in Part_Numbers_Not_Null;
    -- 8.2: Ada Semantics
    Parts_By_Number_Row : in out Parts_Row_Record_Type;
    Parts_By_Number_Status : out Operation_Status);
```

The select procedure below illustrates the use of an **into** clause to specify the name and type of the generated row record parameter.

```
procedure Part_Name_By_Number (
  Input_Pno named Part_Number : Part_Numbers not null) is
  select PNAME named Part_Name
  into Part_Name_By_Number_Row : new Part_Name_Row_Record_Type
  from P
  where PNO = Input_Pno
  status Standard_Map
  ;
```

The above declaration produces the following Ada record type and procedure declarations at the abstract interface.

```
type Part_Name_Row_Record_Type is record -- 8.2: Ada Semantics
  Part_Name : Part_Names_Type;-- 8,2
end record;

procedure Part_Name_By_Number(
  Part_Number      : in Part_Numbers_Not_Null;
  Part_Name_By_Number_Row : in out Part_Name_Row_Record_Type;
  Is_Found          : out boolean);
```

VERSION 2

The example declaration below uses the default **from** clause, which produces a `record_declaration` at the abstract interface.

```
procedure Add_To_Suppliers is
  insert into S (SNO, SNAME, "STATUS", CITY)-- 5.3
  values
  status Operation_Map named Insert_Status
;
```

The `procedure_declaration` above produces the following Ada code at the abstract interface.

```
type Add_To_Suppliers_Row_Type is record -- 5.3: #1
  Sno      : Supplier_Numbers_Type;-- 5.2: Ada Semantics,
  Sname    : Supplier_Names_Type;--Parameter Profiles, #3
  STATUS   : Status_Values_Type;-- 5.8: Ada Semantics
  City     : City_Names_Type;
end record;

procedure Add_To_Suppliers (
  Row      : in Add_To_Suppliers_Row_Type;-- 5.2: Ada Semantics
  Insert_Status : out Operation_Status);
```

This last example illustrates an insert values procedure declaration where all of the values are literals; therefore, no row record parameter is needed for the procedure declaration at the interface.

```
procedure Add_To_Parts is
  insert into P (PNO, PNAME, COLOR, WEIGHT, CITY)
  values ('P02367', 'RIGHT FENDER: TOYOTA', Red, 25,
         'PITTSBURGH')
  status Operation_Map named Insert_Status
;
```

The above declaration produces the following Ada procedure declaration at the abstract interface.

```
procedure Add_To_Parts ( -- 5.8: Ada Semantics, Note
  Insert_Status : out Operation_Status);
```

End Examples

8.10 Value Expressions

A value expression is a value expression of SQL with the following modifications.

1. Enumeration literals, domain parameters and constants are added to the class of expressions.
2. If D is a domain name and VE is a value expression, then the domain conversion $D(VE)$ is a value expression.
3. The syntactic classes parameter specification and variable specification (see 6.2 of ISO 9075:1992) are replaced with the syntactic class parameter (see 8.6 of this International Standard); similarly, the classes parameter name and embedded variable name are replaced with identifier.

Note: This removes the colon from the front of parameter names and eliminates INDICATOR variables.

Four mappings are defined on value_expressions: SQL_{VE} , $AdaNAME$, $DOMAIN$, and $DATACLASS$.

SQL_{VE} maps a value expression to an SQL value expression. It is described in the section "SQL Semantics." below.

The mapping $AdaNAME$ calculates the default names of row record components when value expressions appear in select parameter lists. It is defined as follows:

For any value expression VE , $AdaName(VE)$ is the special value NO_NAME except when VE is an identifier that is the simple name of an input parameter, constant, domain parameter or column, in which case $AdaName(VE) = VE$.

The mapping **DATACLASS** assigns a data class to each well-formed value expression. It is defined as follows.

If VE is an enumeration literal or $DOMAIN(VE)$ is an enumeration domain, then **DATACLASS**(VE) is **enumeration**. Otherwise, **DATACLASS**(VE) is the data type of $SQL_{VE}(VE)$ as defined by ISO 9075:1992 *provided* that data type is not numeric. Otherwise, **DATACLASS**(VE) is one of **integer**, **fixed** or **float** defined as follows:

- a. The **DATACLASS** of a literal, constant, column reference or domain parameter reference is given in sections 5.4, 7.1.4, 7.2 and 7.1.1.1 of this International Standard.
- b. The **DATACLASS** of any numeric value function shall be **integer** (see 6.6 of ISO 9075:1992)
- c. The **DATACLASS**($COUNT(VE')$) shall be **integer**; **DATACLASS**($SUM(VE')$), **DATACLASS**($MIN(VE')$) and **DATACLASS**($MAX(VE')$) shall be **DATACLASS**(VE'). **DATACLASS**($AVG(VE')$) shall be implementation defined (see rule 9 in 6.5 of ISO 9075:1992).
- d. **DATACLASS**($+VE$) and **DATACLASS**($-VE$) is **DATACLASS**(VE). **DATACLASS**($VE_1 \text{ op } VE_2$), for some arithmetic operator, op , shall be the greater of **DATACLASS**(VE_1) and **DATACLASS**(VE_2), where **float** > **fixed** > **integer**.

The DOMAIN mapping and Abstract Syntax Rules of an Expression

The mapping **DOMAIN** assigns a domain to each well formed value expression. A value expression is well formed precisely when the **DOMAIN** mapping assigns it a domain. This section defines the **DOMAIN** mapping and the abstract syntax (type matching rules) of value expressions.

For any value expression, VE , $SQL_{VE}(VE)$ shall conform to the rules of 6.11 ISO 9075:1992. **DOMAIN**(VE) is the special value NO_DOMAIN *except* when defined otherwise by one of the following rules.

1. If VE is an input reference, a reference to a constant, a domain parameter reference or a column reference, then if O is the object referenced, **DOMAIN**(VE) is **DOMAIN**(O);
2. If VE is the set function $MIN(VE')$ or $MAX(VE')$, then **DOMAIN**(VE)=**DOMAIN**(VE') (see 6.5 in ISO 9075:1992);
3. If VE is a character or bit substring function, a fold or a trim function, then **DOMAIN**(VE) is **DOMAIN**(E) where E is the character value expression, bit value expression or trim source directly contained in VE (see 6.7 of ISO 9075:1992);
4. If VE is a case expression, let $\{RE_1, \dots, RE_n\}$ be the set of result expressions of VE . Then
 - $\{RE_1, \dots, RE_n\}$ shall be domain compatible and
 - if for some i , **DOMAIN**(RE_i) $\neq NO_DOMAIN$, then **DOMAIN**(VE)=**DOMAIN**(RE_i) *provided* that, if **DATACLASS**(RE_i) is numeric, then the precision and scale of each result expression, as determined by the rules of ISO 9075:1992, shall not exceed the precision and scale of **DOMAIN**(RE_i) (see 6.9 and 9.3 of ISO 9075:1992);
5. If VE is the cast specification $CAST(VE' \text{ AS } D)$ or the domain conversion $D(VE')$, then D

VERSION 2

shall be a domain name, not a data type and, if D is an enumeration domain VE' shall evaluate to a literal of D . $DOMAIN(VE) = D$;

6. If VE is a string value expression, date time value expression or interval value expression such that the operands of VE are domain compatible and for at least one operand Op , $DOMAIN(Op) \neq NO_DOMAIN$, then $DOMAIN(VE) = DOMAIN(Op)$ (see 6.12, 6.13, 6.14 and 6.15 of ISO 9075:1992).
7. If VE is the numeric value expression $VE_1 \text{ op } VE_2$, let $T_1 = DATACLASS(VE_1)$, $T_2 = DATACLASS(VE_2)$. $D_1 = DOMAIN(VE_1)$, $D_2 = DOMAIN(VE_2)$. VE_1 and VE_2 shall be domain compatible *unless*
 - a. $T_1 = T_2 = \text{fixed}$ and op is either multiplication (*) or division (/), or
 - b. $T_1 = \text{fixed}$, $T_2 = \text{integer}$ and $D_2 = NO_DOMAIN$ and op is either multiplication (*) or division (/), or else
 - c. $T_2 = \text{fixed}$, $T_1 = \text{integer}$ and $D_1 = NO_DOMAIN$ and op is multiplication (*)

If $D_1 \neq NO_DOMAIN$, then $DOMAIN(VE) = D_1$, else $DOMAIN(VE) = D_2$, *unless* rule a applies, in which case $DOMAIN(VE) = NO_DOMAIN$.

Note: These rules simulate Ada's arithmetic restrictions.

8. If VE is a scalar subquery, then $DOMAIN(VE)$ is the domain of the single column in the result of the subquery. See 8.4 of this specification.

SQL Semantics

The SQL value expression derived from a value expression VE is formed by removing all domain conversions and replacing all constants and domain parameters with their values and all enumeration literals with their database representations (see 7.3 of this specification). Let SQL_{VE} represent the function transforming value_expressions into SQL_value_expressions. Let VE be a value_expression. $SQL_{VE}(VE)$ is given recursively as follows:

1. If VE contains no operators, then
 - a. If VE is a column reference or a database literal, then $SQL_{VE}(VE)$ is VE .
 - b. If VE is an enumeration_literal of domain D , and D assigns expression E to that enumeration literal (see rule 7.1.3 of 7.1.3 of this specification), then $SQL_{VE}(VE) = SQL_{VE}(E)$.
 - c. If VE is a reference to the constant whose declaration is given by

```
constant C [: D ] is E ;
```

then $SQL_{VE}(VE) = SQL_{VE}(E)$.
 - d. If VE is a reference to a domain parameter P of domain D , and D assigns the expression E to P (see rule 7.1.3 of 7.1.3 of this specification), then $SQL_{VE}(VE) = SQL_{VE}(E)$.
 - e. If VE is a reference to the input parameter, INP , and $PARM_{SQL}(INP)$ is :C T (for C an SQL_Identifier and T a data type, see 8.6 of this specification), then $SQL_{VE}(VE)$ is

```
:C [INDICATOR :INDIC_NAME(INP)]
```

where $INDICATOR$ $INDIC_NAME(INP)$ appears precisely when $INDIC_{SQL}(INP)$ is defined. See 8.6 of this specification.

VERSION 2

2. If VE is a domain conversion D(VE'), then $SQL_{VE}(VE)$ is

$SQL_{VE}(CAST(VE' AS D))$

If VE is the cast specification

$CAST(VE' AS D)$

then $SQL_{VE}(VE)$ is

- a. $SQL_{VE}(VE')$, if D is an enumeration domain,
 - b. $CAST(SQL_{VE}(VE') AS DBMS_TYPE(D))$ otherwise.
3. Otherwise, $SQL_{VE}(VE)$ is formed by recursively applying the transformer SQL_{VE} to the subexpressions of VE.

8.11 Search Conditions

The concrete syntax of search conditions differs from that of clause 8 of ISO/IEC 9075:1992 only in that

1. Value expressions replace *SQL_Value_Expressions* in the definitions of the atomic predicates;
2. The requirement of comparability in the definitions of the (atomic) *SQL_predicates* is strengthened to require domain compatibility. See 6.4 of this specification.

The conversion transformation SQL_{SC} applied to a search condition produces an *SQL_search_condition* by applying the transformer SQL_{VE} to each value expression in the search condition.

8.12 Status Clauses

A status clause serves to attach a status map to a procedure and optionally rename the status parameter.

```
status_clause ::=  
    status status_reference [named_phrase]
```

Ada Semantics

If a procedure P has a status_clause of the form

```
status M [named Id_1]
```

and the definition of M was given by

```
enumeration T is (L1, ..., Ln);  
status M  
    [named Id_2]  
    uses T  
    is (... , n => L, ...);
```

(see 7.1.8 of this specification),

then:

1. The procedure P_{Ada} (see 8.2 and 8.5 of this specification) shall have a status parameter of type T.
2. The name of the status parameter of P_{Ada} is determined by:

- a. If *Id_1* is present in the *status_clause*, then the name of the status parameter shall be *Id_1*.
- b. If rule (8.12) does not apply, then if *Id_2* is present in the definition of the status map *M*, the name of the status parameter shall be *Id_2* (see 7.1.8 of this specification).
- c. If neither rule (8.12) nor rule (8.12) applies, then the name of the status parameter shall be *Status*.

9 Conformance

9.1 Introduction

This International Standard specifies conforming SAMeDL language and conforming SAMeDL implementations.

Conforming SAMeDL language shall abide by the BNF Format and associated discussion of this specification.

A conforming SAMeDL implementation shall process conforming SAMeDL language according to the Ada Semantics, SQL Semantics and Interface Semantics of this specification.

9.2 Claims of Conformance

9.2.1 Introduction

Claims of conformance to this International Standard shall be to one or more of the following binding styles:

- a) conformance via mapping;
- b) conformance via effects.

9.2.2 Conformance via mapping.

An implementation claiming conformance via mapping shall, for each abstract module, generate an Ada library unit package satisfying the Ada semantics of clause 8.1 of this specification. In addition, an implementation claiming conformance via mapping shall specify one of the following binding substyles:

- i) conformance via SQL module;
- ii) conformance via embedded SQL Syntax.

9.2.2.1 Conformance via SQL module.

An implementation claiming conformance via SQL module shall, for each abstract module, generate an SQL module satisfying the SQL semantics of clause 8.1 of this specification.

9.2.2.2 Conformance via embedded SQL Syntax.

An implementation claiming conformance via embedded SQL Syntax shall, for each abstract module, generate one or more embedded SQL host programs or program fragments, which program(s) or program fragments(s) shall conform to ISO/IEC 9075:1992. Such an implementation shall state which of the host programming languages listed in clause 19 of ISO/IEC 9075:1992 it generates.

9.2.3 Conformance via effects.

Although the processing of abstract modules is defined in terms of the derivation of a program conforming to the Ada programming language standard ISO/IEC 8652:1995 and the SQL Database Language ISO/IEC 9075:1992, implementations of SAMeDL claiming conformance via effects are not constrained to follow that method, provided the same effect is achieved.

9.2.4 Multiple claims of conformance.

An implementation claiming conformance in more than one of the styles of clause 9.2.1 may provide a user option as to which style is to be used.

9.3 Extensions

Conforming SAMeDL implementations shall not process SAMeDL language that is not conforming. Implementation defined facilities may be included into conforming SAMeDL via the extensions facility provided by the SAMeDL (see 4.2.10 of this specification). Schema elements, table elements, statements, query expressions, query specifications, and cursor statements may be extended. The modules, tables, views, cursors, and procedures that contain these extensions are marked (with the keyword **extended**) to indicate that they contain implementation defined functionality.

Implementation defined functionality introduced by extensions may imply an implementation defined extension of the list of reserved words and thereby may prevent proper processing of some texts that otherwise conform to the requirements of this specification.

Annex A SAMeDL_Standard

The predefined SAMeDL definitional module SAMeDL_Standard provides a common location for declarations that are standard for all implementations of the SAMeDL. This definitional module includes the SAMeDL declarations for

- the exceptions SQL_Database_Error and Null_Value_Error (see 6.5 and 7.3 of this specification);
- the SAMeDL standard base domains;
- integer constants whose values are the SQLCODE values predefined in ISO/IEC 9075:1992;
- the domains SQLSTATE_Domain and SQLSTATE_Class_Domain and constants of those domains, mirroring the constants defined in ISO/IEC 9075:1992;
- the pre-defined Status Map Standard_Map.

```

definition module SAMeDL_Standard is
  exception SQL_Database_Error;
  exception Null_Value_Error;

-- SQL_Int Base Domain
  -- SQL_Int is based on the Ada type Interfaces.SQL.Int
  base domain SQL_Int
    (first : integer;
     last  : integer)
  is
    domain pattern is
      'type [self]_Not_Null is new SQL_Int_Not_Null'
      '{ range [first] .. [last]};'
      'type [self]_Type is new SQL_Int;'
      'package [self]_Ops is new SQL_Int_Ops('
      '[self]_Type, [self]_Not_Null);'
    end pattern;
    derived domain pattern is
      'type [self]_Not_Null is new [parent]_Not_Null'
      '{ range [first] .. [last]};'
      'type [self]_Type is new [parent]_Type;'
      'package [self]_Ops is new SQL_Int_Ops('
      '[self]_Type, [self]_Not_Null);'
    end pattern;
    subdomain pattern is
      'subtype [self]_Not_Null is [parent]_Not_Null'
      '{ range [first] .. [last]};'
      'type [self]_Type is new [parent]_Type;'
      'package [self]_Ops is new SQL_Int_Ops('
      '[self]_Type, [self]_Not_Null);'
    end pattern;
    for not null type name use '[self]_Not_Null';
    for null type name use '[self]_Type';
    for data class use integer;
    for dbms type use integer;
    for conversion from dbms to not null use type mark;
    for conversion from not null to null use function
      '[self]_Ops.With_Null';
    for conversion from null to not null use function
      '[self]_Ops.Without_Null';
    for conversion from not null to dbms use type mark;
end SQL_Int;

```

-- SQL_Smallint Base Domain

```

-- SQL_Smallint is based on the Ada type Interfaces.SQL.Smallint
base domain SQL_Smallint
  (first : integer;
   last  : integer)
is
  domain pattern is
    'type [self]_Not_Null is new SQL_Smallint_Not_Null'
    '{ range [first] .. [last]};'
    'type [self]_Type is new SQL_Smallint;'
    'package [self]_Ops is new SQL_Smallint_Ops('
      '[self]_Type, [self]_Not_Null);'
  end pattern;
  derived domain pattern is
    'type [self]_Not_Null is new [parent]_Not_Null'
    '{ range [first] .. [last]};'
    'type [self]_Type is new [parent]_Type;'
    'package [self]_Ops is new SQL_Smallint_Ops('
      '[self]_Type, [self]_Not_Null);'
  end pattern;
  subdomain pattern is
    'subtype [self]_Not_Null is [parent]_Not_Null'
    '{ range [first] .. [last]};'
    'type [self]_Type is new [parent]_Type;'
    'package [self]_Ops is new SQL_Smallint_Ops('
      '[self]_Type, [self]_Not_Null);'
  end pattern;
  for not null type name use '[self]_Not_Null';
  for null type name use '[self]_Type';
  for data class use integer;
  for dbms type use smallint;
  for conversion from dbms to not null use type mark;
  for conversion from not null to null use function
    '[self]_Ops.With_Null';
  for conversion from null to not null use function
    '[self]_Ops.Without_Null';
  for conversion from not null to dbms use type mark;
end SQL_Smallint;

```

-- SQL_Real Base Domain

```

-- SQL_Real is based on the Ada type Interfaces.SQL.Real
base domain SQL_Real
  (first : float;
   last  : float)
is
  domain pattern is
    'type [self]_Not_Null is new SQL_Real_Not_Null'
    '{ range [first] .. [last]};'
    'type [self]_Type is new SQL_Real;'
    'package [self]_Ops is new SQL_Real_Ops('
      '[self]_Type, [self]_Not_Null);'
  end pattern;
  derived domain pattern is
    'type [self]_Not_Null is new [parent]_Not_Null'
    '{ range [first] .. [last]};'
    'type [self]_Type is new [parent]_Type;'
    'package [self]_Ops is new SQL_Real_Ops('
      '[self]_Type, [self]_Not_Null);'
  end pattern;
  subdomain pattern is
    'subtype [self]_Not_Null is [parent]_Not_Null'
    '{ range [first] .. [last]};'
    'type [self]_Type is new [parent]_Type;'
    'package [self]_Ops is new SQL_Real_Ops('

```

VERSION 2

```
        '[self]_Type, [self]_Not_Null);'
    end pattern;
    for not null type name use '[self]_Not_Null';
    for null type name use '[self]_Type';
    for data class use float;
    for dbms type use real;
    for conversion from dbms to not null use type mark;
    for conversion from not null to null use function
        '[self]_Ops.With_Null';
    for conversion from null to not null use function
        '[self]_Ops.Without_Null';
    for conversion from not null to dbms use type mark;
end SQL_Real;
```

-- SQL_Double_Precision Base Domain

```
-- SQL_Double_Precision is based on the Ada type
-- Interfaces.SQL.Double_Precision
base domain SQL_Double_Precision
    (first : float;
     last  : float)
is
    domain pattern is
        'type [self]_Not_Null is new SQL_Double_Precision_Not_Null'
        '{ range [first] .. [last]};'
        'type [self]_Type is new SQL_Double_Precision;'
        'package [self]_Ops is new SQL_Double_Precision_Ops('
        '[self]_Type, [self]_Not_Null);'
    end pattern;
    derived domain pattern is
        'type [self]_Not_Null is new [parent]_Not_Null'
        '{ range [first] .. [last]};'
        'type [self]_Type is new [parent]_Type;'
        'package [self]_Ops is new SQL_Double_Precision_Ops('
        '[self]_Type, [self]_Not_Null);'
    end pattern;
    subdomain pattern is
        'subtype [self]_Not_Null is [parent]_Not_Null'
        '{ range [first] .. [last]};'
        'type [self]_Type is new [parent]_Type;'
        'package [self]_Ops is new SQL_Double_Precision_Ops('
        '[self]_Type, [self]_Not_Null);'
    end pattern;
    for not null type name use '[self]_Not_Null';
    for null type name use '[self]_Type';
    for data class use float;
    for dbms type use double precision;
    for conversion from dbms to not null use type mark;
    for conversion from not null to null use function
        '[self]_Ops.With_Null';
    for conversion from null to not null use function
        '[self]_Ops.Without_Null';
    for conversion from not null to dbms use type mark;
end SQL_Double_Precision;
```

-- SQL_Numeric Base Domain

```
-- SQL_Numeric is based on the Ada types
-- in the package Interfaces.SQL.Numerics
base domain SQL_Numeric
    (first : numeric;
     last  : numeric;
     digits : integer)
is
    domain pattern is
        'type [self]_Not_Null is new'
        'Interfaces.SQL.Numerics.Scale_[scale] digits [digits];'
```

```

    'package [self]_Pkg is new SQL_Numeric_Pkg('
      'Scale_Type => [self]_not_null);'
    'type [self]_type is new [self]_Pkg.SQL_Numeric;'
    'package [self]_Ops is new [self]_Pkg.SQL_Numeric_Ops('
      '[self]_Type, [self]_Not_Null);'
  end pattern;
  derived domain pattern is
    'type [self]_Not_Null is new [parent]_Not_Null'
      '{ range [first] .. [last]};'
    'type [self]_Type is new [parent]_Type;'
    'package [self]_Ops is new [self]_Pkg.SQL_Numeric_Ops('
      '[self]_Type, [self]_Not_Null);'
  end pattern;
  subdomain pattern is
    'subtype [self]_Not_Null is [parent]_Not_Null'
      '{ range [first] .. [last]};'
    'type [self]_Type is new [parent]_Type;'
    'package [self]_Ops is new [self]_Pkg.SQL_Numeric_Ops('
      '[self]_Type, [self]_Not_Null);'
  end pattern;
  for not null type name use '[self]_Not_Null';
  for null type name use '[self]_Type';
  for data class use numeric;
  for dbms type use 'numeric ([digits], [scale])';
  for conversion from dbms to not null use type mark;
  for conversion from not null to null use function
    '[self]_Ops.With_Null';
  for conversion from null to not null use function
    '[self]_Ops.Without_Null';
  for conversion from not null to dbms use type mark;
end SQL_Numeric;

```

-- SQL_Char Base Domain

```

-- SQL_Char is based on the Ada type Interfaces.SQL.Char
base domain SQL_Char
is
  domain pattern is
    'type [self]NN_Base is new SQL_Char_Not_Null;'
    'subtype [self]_Not_Null is [self]NN_Base (1 .. [length]);'
    'type [self]_Base is new SQL_Char;'
    'subtype [self]_Type is [self]_Base ('
      '[self]_Not_Null''length);'
    'package [self]_Ops is new SQL_Char_Ops('
      '[self]_Base, [self]NN_Base);'
  end pattern;
  derived domain pattern is
    'type [self]NN_Base is new [parent]NN_Base;'
    'subtype [self]_Not_Null is [self]NN_Base (1 .. [length]);'
    'type [self]_Base is new [parent]_Base;'
    'subtype [self]_Type is [self]_Base ('
      '[self]_Not_Null''length);'
    'package [self]_Ops is new SQL_Char_Ops ('
      '[self]_Base, [self]NN_Base);'
  end pattern;
  subdomain pattern is
    'subtype [self]NN_Base is [parent]NN_Base;'
    'subtype [self]_Not_Null is [parent]NN_Base (1 .. [length]);'
    'subtype [self]_Base is new [parent]_Base;'
    'subtype [self]_Type is [self]_Base ('
      '[self]_Not_Null''length);'
    'package [self]_Ops is new SQL_Char_Ops ('
      '[self]_Base, [self]NN_Base);'
  end pattern;
  for not null type name use '[self]_Not_Null';
  for null type name use '[self]_Type';

```

VERSION 2

```
for data class use character;
for dbms type use character '([length]]';
for conversion from dbms to not null use type mark;
for conversion from not null to null use function
  '[self]_Ops.With_Null';
for conversion from null to not null use function
  '[self]_Ops.Without_Null';
for conversion from not null to dbms use type mark;
end SQL_Char;
```

-- SQL_NChar Base Domain

```
-- SQL_NChar is identical to SQL_CHAR except that it is
-- based on the Ada type Interfaces.SQL.NChar
base domain SQL_NChar
is
  domain pattern is
    'type [self]NN_Base is new SQL_NChar_Not_Null;'
    'subtype [self]_Not_Null is [self]NN_Base (1 .. [length]);'
    'type [self]_Base is new SQL_NChar;'
    'subtype [self]_Type is [self]_Base ('
      '[self]_Not_Null''length);'
    'package [self]_Ops is new SQL_NChar_Ops('
      '[self]_Base, [self]NN_Base);'
  end pattern;
  derived domain pattern is
    'type [self]NN_Base is new [parent]NN_Base;'
    'subtype [self]_Not_Null is [self]NN_Base (1 .. [length]);'
    'type [self]_Base is new [parent]_Base;'
    'subtype [self]_Type is [self]_Base ('
      '[self]_Not_Null''length);'
    'package [self]_Ops is new SQL_NChar_Ops ('
      '[self]_Base, [self]NN_Base);'
  end pattern;
  subdomain pattern is
    'subtype [self]NN_Base is [parent]NN_Base;'
    'subtype [self]_Not_Null is [parent]NN_Base (1 .. [length]);'
    'subtype [self]_Base is new [parent]_Base;'
    'subtype [self]_Type is [self]_Base ('
      '[self]_Not_Null''length);'
    'package [self]_Ops is new SQL_NChar_Ops ('
      '[self]_Base, [self]NN_Base);'
  end pattern;
  for not null type name use '[self]_Not_Null';
  for null type name use '[self]_Type';
  for data class use character;
  for dbms type use character '([length]]';
  for conversion from dbms to not null use type mark;
  for conversion from not null to null use function
    '[self]_Ops.With_Null';
  for conversion from null to not null use function
    '[self]_Ops.Without_Null';
  for conversion from not null to dbms use type mark;
end SQL_NChar;
```

-- SQL_VarChar Base Domain

```
base domain SQL_VarChar
is
  domain pattern is
    'type [self]_Not_Null is new SQL_VarChar_Not_Null'
    'type [self]_Type is new SQL_VarChar;'
    'package [self]_Ops is new SQL_VarChar_Ops('
      '[self]_Type, [self]_Not_Null);'
  end pattern;
  derived domain pattern is
    'type [self]_Not_Null is new [parent]_Not_Null;'
    'type [self]_Type is new [parent]_Type;
```

```

        'package [self]_Ops is new SQL_VarChar_Ops (
          '[self]_Type, [self]_Not_Null);'
      end pattern;
      subdomain pattern is
        'subtype [self]_Not_Null is [parent]_Not_Null';
        'subtype [self]_Type is [parent]_Type ('
        'package [self]_Ops is new SQL_VarChar_Ops (
          '[self]_Type, [self]_Not_Null);'
      end pattern;
      for data class use character;
      for dbms type use character varying '([length]]';
      for conversion from dbms to not null use type mark;
      for conversion from not null to null use function
        '[self]_Ops.With_Null';
      for conversion from null to not null use function
        '[self]_Ops.Without_Null';
      for conversion from not null to dbms use type mark;
    end SQL_VarChar;

```

-- SQL_Bit Base Domain

```

-- SQL_Bit is similar to the character string base domains
base domain SQL_Bit
is
  domain pattern is
    'type [self]NN_Base is new SQL_Bit_Not_Null';
    'subtype [self]_Not_Null is [self]NN_Base (1 .. [length]);'
    'type [self]_Base is new SQL_Bit';
    'subtype [self]_Type is [self]_Base ('
      '[self]_Not_Null''length);'
    'package [self]_Ops is new SQL_Bit_Ops('
      '[self]_Base, [self]NN_Base);'
  end pattern;
  derived domain pattern is
    'type [self]NN_Base is new [parent]NN_Base';
    'subtype [self]_Not_Null is [self]NN_Base (1 .. [length]);'
    'type [self]_Base is new [parent]_Base';
    'subtype [self]_Type is [self]_Base ('
      '[self]_Not_Null''length);'
    'package [self]_Ops is new SQL_Bit_Ops ('
      '[self]_Base, [self]NN_Base);'
  end pattern;
  subdomain pattern is
    'subtype [self]NN_Base is [parent]NN_Base';
    'subtype [self]_Not_Null is [parent]NN_Base (1 .. [length]);'
    'subtype [self]_Base is new [parent]_Base';
    'subtype [self]_Type is [self]_Base ('
      '[self]_Not_Null''length);'
    'package [self]_Ops is new SQL_Bit_Ops ('
      '[self]_Base, [self]NN_Base);'
  end pattern;
  for not null type name use '[self]_Not_Null';
  for null type name use '[self]_Type';
  for data class use bit
  for dbms type use bit '([length]]';
  for conversion from dbms to not null use type mark;
  for conversion from not null to null use function
    '[self]_Ops.With_Null';
  for conversion from null to not null use function
    '[self]_Ops.Without_Null';
  for conversion from not null to dbms use type mark;
end SQL_Bit;

```

-- SQL_Enumeration_As_Int Base Domain

```

-- SQL_Enumeration_As_Int is based on the Ada type
-- Interfaces.SQL.Int
base domain SQL_Enumeration_As_Int

```

```

    (map := pos)
is
    domain pattern is
        'type [self]_Not_Null is new [enumeration];'
        'package [self]_Pkg is new SQL_Enumeration_Pkg('
            '[enumeration]);'
        'type [self]_Type is new [self]_Pkg.SQL_Enumeration;'
    end pattern;
    derived domain pattern is
        'type [self]_Not_Null is new [parent]_Not_Null;'
        'type [self]_Type is new [parent]_Type;'
    end pattern;
    subdomain pattern is
        'subtype [self]_Not_Null is [parent]_Not_Null;'
        'subtype [self]_Type is [parent]_Type;'
    end pattern;

```

-- SQL_Enumeration_As_Char Base Domain

```

-- SQL_Enumeration_As_Char is based on the Ada type
-- Interfaces.SQL.Char
base domain SQL_Enumeration_As_Char
    (map := image)
is
    domain pattern is
        'type [self]_Not_Null is new [enumeration];'
        'package [self]_Pkg is new SQL_Enumeration_Pkg('
            '[enumeration]);'
        'type [self]_Type is new [self]_Pkg.SQL_Enumeration;'
    end pattern;
    derived domain pattern is
        'type [self]_Not_Null is new [parent]_Not_Null;'
        'type [self]_Type is new [parent]_Type;'
    end pattern;
    subdomain pattern is
        'subtype [self]_Not_Null is [parent]_Not_Null;'
        'subtype [self]_Type is [parent]_Type;'
    end pattern;
    for not null type name use '[self]_Not_Null';
    for null type name use '[self]_Type';
    for data class use enumeration;
    for dbms type use character '([length])';
    for conversion from not null to null use function
        '[self]_Pkg.With_Null';
    for conversion from null to not null use function
        '[self]_Pkg.Without_Null';
end SQL_Enumeration_As_Char;

```

-- Temporal Base Domains

```

base domain SQL_Date is
    for data class use date;
    for dbmstype use none;
end SQL_Date;
base domain SQL_Time is
    for data class use time;
    for dbmstype use none;
end SQL_Time;
base domain SQL_Timestamp is
    for data class use timestamp;
    for dbmstype use none;
end SQL_Timestamp;
base domain SQL_Interval is
    for data class use interval;
    for dbmstype use none;
end SQL_Interval;

```

-- SQLCODE Standardvalues

```

-- SQLCODE for successful execution of a statement
constant Success is 0;
-- SQLCODE for data not found
constant Not_Found is 100;
-- SQLSTATE domains and constants
-- Domain declarations for SQLSTATE values
domain SQLSTATE_Domain is new SQL_Char not null
  (Length => 5);
domain SQLSTATE_Class_Domain is new SQL_Char not null
  (Length => 2);
-- These SQLSTATE values and their names
-- are taken from the definition of the Ada package Interfaces.SQL,
-- defined in 12.4 of ISO/IEC 9075:1992.
-- Any discrepancy between this list and the list in the citation
-- is resolved in favor of the later, which is definitive.
constant Ambiguous_Cursor_Name : SQLSTATE_Class_Domain is '3C';
constant Ambiguous_Cursor_Name_No_Subclass :
  SQLSTATE_Domain is '3C000';
constant Cardinality_Violation : SQLSTATE_Class_Domain is '21';
constant Cardinality_Violation_No_Subclass :
  SQLSTATE_Domain is '21000';
constant Connection_Exception : SQLSTATE_Class_Domain is '08';
constant Connection_Exception_No_Subclass :
  SQLSTATE_Domain is '08000';
constant Connection_Exception_Connection_Does_Not_Exist :
  SQLSTATE_Domain is '08003';
constant Connection_Exception_Connection_Failure :
  SQLSTATE_Domain is '08006';
constant Connection_Exception_Connection_Name_In_Use :
  SQLSTATE_Domain is '08002';

constant Connection_Exception_SQLClient_Unable_To_Establish_SQLConnection :
  SQLSTATE_Domain is '08001';

constant Connection_Exception_SQLServer_Rejected_Establishment_Of_SQLConne
tion :
  SQLSTATE_Domain is '08004';
constant Connection_Exception_Transaction_Resolution_Unknown :
  SQLSTATE_Domain is '08007';
constant Data_Exception : SQLSTATE_Class_Domain is '22';
constant Data_Exception_No_Subclass :
  SQLSTATE_Domain is '22000';
constant Data_Exception_Character_Not_in_Repertoire :
  SQLSTATE_Domain is '22008';
constant Data_Exception_DateTime_Field_Overflow :
  SQLSTATE_Domain is '22008';
constant Data_Exception_Division_By_Zero :
  SQLSTATE_Domain is '22012';
constant Data_Exception_Error_In_Assignment :
  SQLSTATE_Domain is '22005';
constant Data_Exception_Indicator_Overflow :
  SQLSTATE_Domain is '22022';
constant Data_Exception_Interval_Field_Overflow :
  SQLSTATE_Domain is '22015';
constant Data_Exception_Invalid_Character_Value_For_Cast :
  SQLSTATE_Domain is '22018';
constant Data_Exception_Invalid_DateTime_Format :
  SQLSTATE_Domain is '22007';
constant Data_Exception_Invalid_Escape_Character :
  SQLSTATE_Domain is '22019';
constant Data_Exception_Invalid_Escape_Sequence :
  SQLSTATE_Domain is '22025';
constant Data_Exception_Invalid_Parameter_Value :
  SQLSTATE_Domain is '22023';
constant Data_Exception_Invalid_Time_Zone_Displacement_Value :
  SQLSTATE_Domain is '22009';

```

VERSION 2

```
constant Data_Exception_Null_Value_No_Indicator_Parameter :
    SQLSTATE_Domain is '22002';
constant Data_Exception_Numeric_Value_Out_of_Range :
    SQLSTATE_Domain is '22003';
constant Data_Exception_String_Data_Length_Mismatch :
    SQLSTATE_Domain is '22026';
constant Data_Exception_String_Data_Right_Truncation :
    SQLSTATE_Domain is '22001';
constant Data_Exception_Substring_Error :
    SQLSTATE_Domain is '22011';
constant Data_Exception_Trim_Error :
    SQLSTATE_Domain is '22027';
constant Data_Exception_Unterminated_C_String :
    SQLSTATE_Domain is '22024';
constant Dependent_Privilege_Descriptors_Still_Exist :
    SQLSTATE_Class_Domain is '2B';
constant Dependent_Privilege_Descriptors_Still_Exist_No_Subclass :
    SQLSTATE_Domain is '2B000';
constant Dynamic_SQL_Error : SQLSTATE_Class_Domain is '07';
constant Dynamic_SQL_Error_No_Subclass :
    SQLSTATE_Domain is '07000';
constant Dynamic_SQL_Error_Cursor_Specification_Cannot_Be_Executed :
    SQLSTATE_Domain is '07003';
constant Dynamic_SQL_Error_Invalid_Descriptor_Count :
    SQLSTATE_Domain is '07008';
constant Dynamic_SQL_Error_Invalid_Descriptor_Index :
    SQLSTATE_Domain is '07009';
constant
Dynamic_SQL_Error_Prepared_Statement_Not_A_Cursor_Specification :
    SQLSTATE_Domain is '07005';
constant Dynamic_SQL_Error_Restricted_Data_Type_Attribute_Violation :
    SQLSTATE_Domain is '07006';
constant
Dynamic_SQL_Error_Using_Clause_Does_Not_Match_Dynamic_Parameter_Sec :
    SQLSTATE_Domain is '07001';
constant Dynamic_SQL_Error_Using_Clause_Does_Not_Match_Target_Spec :
    SQLSTATE_Domain is '07002';
constant
Dynamic_SQL_Error_Using_Clause_Required_For_Dynamic_Parameters :
    SQLSTATE_Domain is '07004';
constant Dynamic_SQL_Error_Using_Clause_Required_For_Result_Fields :
    SQLSTATE_Domain is '07007';
constant Feature_Not_Supported : SQLSTATE_Class_Domain is '0A';
constant Feature_Not_Supported_No_Subclass :
    SQLSTATE_Domain is '0A000';
constant Feature_Not_Supported_Mutliple_Environment_Transactions :
    SQLSTATE_Domain is '0A001';
constant Integrity_Constraint_Violation : SQLSTATE_Class_Domain is '23';
constant Integrity_Constraint_Violation_No_Subclass :
    SQLSTATE_Domain is '23000';
constant
Invalid_Authorization_Specification : SQLSTATE_Class_Domain is '2';
constant Invalid_Authorization_Specification_No_Subclass :
    SQLSTATE_Domain is '28000';
constant Invalid_Catalog_Name : SQLSTATE_Class_Domain is '3D';
constant Invalid_Catalog_Name_No_Subclass :
    SQLSTATE_Domain is '3D000';
constant Invalid_Character_Set_Name : SQLSTATE_Class_Domain is '2C';
constant Invalid_Character_Set_Name_No_Subclass :
    SQLSTATE_Domain is '2C000';
constant Invalid_Condition_Number : SQLSTATE_Class_Domain is '35';
constant Invalid_Condition_Number_No_Subclass :
    SQLSTATE_Domain is '35000';
constant Invalid_Connection_Name : SQLSTATE_Class_Domain is '2E';
constant Invalid_Connection_Name_No_Subclass :
    SQLSTATE_Domain is '2E000';
constant Invalid_Cursor_Name : SQLSTATE_Class_Domain is '34';
constant Invalid_Cursor_Name_No_Subclass :
```

```

    SQLSTATE_Domain is '34000';
constant Invalid_Cursor_State : SQLSTATE_Class_Domain is '24';
constant Invalid_Cursor_State_No_Subclass :
    SQLSTATE_Domain is '24000';
constant Invalid_Schema_Name : SQLSTATE_Class_Domain is '3F';
constant Invalid_Schema_Name_No_Subclass :
    SQLSTATE_Domain is '3F000';
constant Invalid_SQL_Descriptor_Name : SQLSTATE_Class_Domain is '33';
constant Invalid_SQL_Descriptor_Name_No_Subclass :
    SQLSTATE_Domain is '33000';
constant Invalid_SQL_Statement_Name : SQLSTATE_Class_Domain is '26';
constant Invalid_SQL_Statement_Name_No_Subclass :
    SQLSTATE_Domain is '26000';
constant Invalid_Transaction_State : SQLSTATE_Class_Domain is '25';
constant Invalid_Transaction_State_No_Subclass :
    SQLSTATE_Domain is '25000';
constant
Invalid_Transaction_Termination : SQLSTATE_Class_Domain is '2D';
constant Invalid_Transaction_Termination_No_Subclass :
    SQLSTATE_Domain is '2D000';
constant No_Data : SQLSTATE_Class_Domain is '02';
constant No_Data_No_Subclass :
    SQLSTATE_Domain is '02000';
constant Remote_Database_Access : SQLSTATE_Class_Domain is 'HZ';
constant Remote_Database_Access_No_Subclass :
    SQLSTATE_Domain is 'HZ000';
constant Successful_Completion : SQLSTATE_Class_Domain is '00';
constant Successful_Completion_No_Subclass :
    SQLSTATE_Domain is '00000';
constant
Syntax_Error_Or_Access_Rule_Violation : SQLSTATE_Class_Domain is '42';
constant Syntax_Error_Or_Access_Rule_Violation_No_Subclass :
    SQLSTATE_Domain is '42000';
constant Syntax_Error_Or_Access_Rule_Violation_In_Direct_Statement :
    SQLSTATE_Class_Domain is '2A';
constant
Syntax_Error_Or_Access_Rule_Violation_In_Direct_Statement_No_Subclass :
    SQLSTATE_Domain is '2A000';
constant Syntax_Error_Or_Access_Rule_Violation_In_Dynamic_Statement :
    SQLSTATE_Class_Domain is '37';
constant
Syntax_Error_Or_Access_Rule_Violation_In_Dynamic_Statement_No_Subclass :
    SQLSTATE_Domain is '37000';
constant Transaction_Rollback : SQLSTATE_Class_Domain is '40';
constant Transaction_Rollback_No_Subclass :
    SQLSTATE_Domain is '40000';
constant Transaction_Rollback_Integrity_Constraint_Violation :
    SQLSTATE_Domain is '40002';
constant Transaction_Rollback_Serialization_Failure :
    SQLSTATE_Domain is '40001';
constant Transaction_Rollback_Statement_Completion_Unknown :
    SQLSTATE_Domain is '40003';
constant
Triggered_Data_Change_Violation : SQLSTATE_Class_Domain is '27';
constant Triggered_Data_Change_Violation_No_Subclass :
    SQLSTATE_Domain is '27000';
constant Warning : SQLSTATE_Class_Domain is '01';
constant Warning_No_Subclass :
    SQLSTATE_Domain is '01000';
constant Warning_Cursor_Operation_Conflict :
    SQLSTATE_Domain is '01001';
constant Warning_Disconnect_Error :
    SQLSTATE_Domain is '01002';
constant Warning_Implicit_Zero_Bit_Padding :
    SQLSTATE_Domain is '01008';
constant Warning_Insufficient_Item_Descriptor_Areas :
    SQLSTATE_Domain is '01005';
constant Warning_Null_Value_Eliminated_in_Set_Function :

```

VERSION 2

```
    SQLSTATE_Domain is '01003';
constant Warning_Privilege_Not_Granted :
    SQLSTATE_Domain is '01007';
constant Warning_Privilege_Not_Revoked :
    SQLSTATE_Domain is '01006';
constant Warning_Query_Expression_Too_Long_For_Information_Schema :
    SQLSTATE_Domain is '0100A';
constant Warning_Search_Condition_Too_Long_For_Information_Schema :
    SQLSTATE_Domain is '01009';
constant Warning_String_Data_Right_Truncation_Warning :
    SQLSTATE_Domain is '01004';
constant With_Check_Option_Violation : SQLSTATE_Class_Domain is '44';
constant With_Check_Option_Violation_No_Subclass :
    SQLSTATE_Domain is '44000';
sqlstate status Standard_Map
    named Is_Found
    uses boolean
is
    (Successful_Completion_No_Subclass => True,
     No_Data_No_Subclass => False);
end SAMeDL_Standard;
```

Annex B SAMeDL_System

The predefined SAMeDL definitional module SAMeDL_System provides a common location for the declaration of implementation-defined constants that are specific to a particular DBMS/Ada compiler platform.

```

with SAMeDL_Standard; use SAMeDL_Standard;
definition module SAMeDL_System is
  -- Smallest (most negative) value of any integer type
  constant Min_Int is implementation defined;
  -- Largest (most positive) value of any integer type
  constant Max_Int is implementation defined;
  -- Smallest value of any SQL_Int type
  constant Min_SQL_Int is implementation defined;
  -- Largest value of any SQL_Int type
  constant Max_SQL_Int is implementation defined;
  -- Smallest value of any SQL_Smallint type
  constant Min_SQL_Smallint is implementation defined;
  -- Largest value of any SQL_Smallint type
  constant Max_SQL_Smallint is implementation defined;
  -- Largest value allowed for the number of significant decimal
  -- digits in any floating point constraint
  constant Max_Digits is implementation defined;
  -- Largest value allowed for the number of significant decimal
  -- digits in an SQL_Real floating point constraint
  constant SQL_Real_Digits is implementation defined;
  -- Largest value allowed for the number of significant decimal
  -- digits in an SQL_Double_Precision floating point constraint
  constant SQL_Double_Precision_Digits is implementation defined;
  -- Largest value allowed for the number of characters in a
  -- character string constraint
  constant Max_SQL_Char_Length is implementation defined;
  -- implementation defined SQLSTATE and SQLCODE values
  -- should be declared as constants here
end SAMeDL_System;

```

Annex C Standard Support Operations and Specifications

The following two sections discuss the SAME standard support packages. The first section describes how they support the standard base domains, and the second section lists their Ada package specifications.

C.1 Standard Base Domain Operations

The SAME standard support packages encapsulate the Ada type definitions of the standard base domains, as well as the operations that provide the data semantics for domains declared using these base domains. This section describes the nature of the support packages, namely the Ada data types and the operations on objects of these types.

The SQL standard package `Interfaces.SQL` (see C in this International Standard and 12.3.8.a.iii of ISO/IEC 9075:1992; contains the type definitions for a DBMS platform that define the Ada representations of the concrete SQL data types. A standard base domain exists in the SAMeDL for each type in `Interfaces.SQL` (except for `SQLCode_Type`), and these base domains are each supported by one of the SAME standard support packages. In addition to the above base domains, two standard base domains exist that provide data semantics for Ada enumeration types.

Each support package defines a not null-bearing and a null-bearing type for the base domain. The not null-bearing type is a visible Ada type derived from the corresponding type in `Interfaces.SQL` with no added constraints. This type provides the Ada application programmer with Ada data semantics for data in the database. The null-bearing type is an Ada limited private type used to support data semantics of the SQL null value. In particular, the null-bearing type may contain the null value; the not null-bearing type may not.

Domains are derived from base domains by the declaration of two Ada data types, derived from the types in the support packages, and the instantiation of the generic operations package with these types. The type derivations and the package instantiation provide the domain with the complete set of operations that define the data semantics for that domain. These operations are described below, grouped by data class.

C.1.1 All Domains

All domains derived from the standard base domains make an *Assign* procedure available to the application because the type that supports the SQL data semantics is an Ada limited private type. For the numeric domains, this procedure enforces the range constraints that are specified for the domain when it is declared. The Ada *Constraint_Error* exception is raised by these procedures if the value to be assigned falls outside of the specified range.

A parameterless function named *Null_SQL_<type>* is available for all domains as well. This function returns an object of the null-bearing type of the appropriate domain whose value is the SQL null value.

Every domain has a set of conversion functions available for converting between the not null-bearing type and the null-bearing type. The function *With_Null* converts an object of the not null-bearing type to an object

of the null-bearing type. The function *Without_Null* converts an object of the null-bearing type to an object of the not null-bearing type. *Without_Null* will raise the *Null_Value_Error* exception if the value of the object that it is converting is the SQL null value, since an object of the not null-bearing type can never be null.

Two testing functions are available for each domain as well. The boolean functions *Is_Null* and *Not_Null* test objects of the null-bearing type, returning the appropriate boolean value indicating whether or not an object contains the SQL null value.

Additionally, all domains provide two sets of comparison operators that operate on objects of the null-bearing type. The first set of operators returns boolean values, and the second set of operators returns objects of the type *Boolean_With_Unknown*, defined in the support package *SQL_Boolean_Pkg* (see Section C), which implements three-valued logic (see @cite(guide)). The boolean comparison operators are *=*, */=*, *<*, *>*, *<=*, and *>=*, and return the value *False* if either of the objects contains the SQL null value. @Foot[Note: These semantics have a peculiar side effect, namely that, for objects O_1 and O_2 , the boolean expression $(O_1 < O_2)$ or else $(O_1 >= O_2)$ is not a tautology.] Otherwise, these operators perform the comparison, and return the appropriate boolean result. The *Boolean_With_Unknown* comparison operators are *Equals* and *Not_Equals*, *<*, *>*, *<=*, and *>=*, and return the value *Unknown* if either of the objects contains the SQL null value. Otherwise, these operators perform the comparison, and return the *Boolean_With_Unknown* values *True* or *False*.

C.1.2 Numeric Domains

In addition to the operations mentioned above, all numeric domains provide unary and binary arithmetic operations for the null-bearing type of the domain. The subprograms that implement these operations provide the data semantics of the SQL null value with respect to these arithmetic operations. Specifically, any arithmetic operation applied to a null value results in the null value. Otherwise, the operation is defined to be the same as the Ada operation. The unary operations that are provided are *+*, *-*, and *Abs*. The binary operations include *+*, *-*, ***, and */*. Finally, all numeric domains provide the exponentiation operation (****).

C.1.3 Int and Smallint Domains

Int and *Smallint* domains provide the application programmer with the Ada functions *Mod* and *Rem* that operate on objects of the null-bearing type. Again, the subprograms that implement these operations provide the data semantics of the SQL null value with respect to these arithmetic operations. As with the other arithmetic operations, *Mod* and *Rem* return the null value when applied to an object containing the null value. Otherwise, they are defined to be the same as the Ada operation.

These domains also make *Image* and *Value* functions available to the application programmer. Both of these functions are overloaded, meaning that there are *Image* and *Value* functions that operate on objects of both the not null-bearing and the null-bearing types of the domain. The *Image* function converts an object of an *Int* or *Smallint* domain to a character representation of the integer value. The *Value* function converts a character representation of an integer value to an object of an *Int* or *Smallint* domain. These functions perform the same operation as the Ada attribute functions of the same name, except that the character set of the character inputs and outputs is that of the underlying Interfaces.SQL.Char character set. If the *Image* and *Value* functions are applied to objects of the null-bearing type containing the null value, a null character object and a null integer object are returned respectively.

C.1.4 Character Domains

In addition to the operations provided by all domains, character domains provide the application programmer with some string manipulation and string conversion operations.

Character domains provide two string manipulation functions that operate on objects of the null-bearing type. The first one is the catenation function (&). If either of the input character objects contains the null value, then the object returned contains the null value. Otherwise this operation is the same as the Ada catenation operation. The other function is the *Substring* function, which is patterned after the substring function in ISO/IEC 9075:1992. This function returns the portion of the input character object specified by the *Start* and *Length* index inputs. An Ada *Constraint_Error* is raised if the substring specification is not contained entirely within the input string.

The remaining operations provided by the character domains are conversion functions. A *To_String* and a *To_Unpadded_String* function exist for both the not null-bearing and the null-bearing types of the domain. The *To_String* function converts its input, which exists as an object whose value is comprised of characters from the underlying character set of the platform, to an object of the Ada predefined type *Standard.String*. If conversion of a null-bearing object containing the null value is attempted, the *Null_Value_Error* exception is raised. The *To_Unpadded_String* functions are identical in every way to the *To_String* functions, except that trailing blanks are stripped from the value.

The *Without_Null_Unpadded* function is identical to the *Without_Null* function, described in section C above, except that trailing blanks are stripped from the value.

Two functions exist that convert objects of the Ada predefined type *Standard.String* to objects of the not null-bearing and null-bearing types of the domain. The *To_SQL_Char_Not_Null* function converts an object of type *Standard.String* to the not null-bearing type of the domain. The *To_SQL_Char* function converts an object of type *Standard.String* to an object of the null-bearing type.

Finally, character domains provide the function *Unpadded_Length*, which returns the length of the character string representation without trailing blanks. This function operates on objects of the null-bearing type, and raises the *Null_Value_Error* exception if the input object contains the null value.

C.1.5 Enumeration Domains

Enumeration domains provide functions for the null-bearing type that are normally available as Ada attribute functions for the not null-bearing type. The *Image* and *Value* functions have the same semantics as described for *Int* and *Smallint* domains in Section C above, except that they operate on enumeration values rather than integers.

The *Pred* and *Succ* functions operate on objects of the null-bearing type, and return the previous and next enumeration literals of the underlying enumeration type, respectively. If these functions are applied to objects containing the null value, an object containing the null value is returned.

VERSION 2

The last two functions are the *Pos* and *Val* functions. These functions also operate on objects of the null-bearing type. *Pos* returns a value of the Ada predefined type *Standard.Integer* representing the position (relative to zero) of the enumeration literal that is the value of the input object. If the input object contains the null value, then the *Null_Value_Error* exception is raised. The *Val* function accepts a value of the predefined type *Standard.Integer* and returns the enumeration literal whose position in the underlying enumeration type is specified by that value. If the input integer value falls outside the range of available enumeration literals, the Ada *Constraint_Error* is raised.

C.1.6 Boolean Functions

The SAME standard support package *SQL_Boolean_Pkg* defines a number of boolean functions, namely *not*, *and*, *or*, and *xor*, which implement three-valued logic as defined in ISO/IEC 9075:1992. All of these functions operate on two input parameters of the type *Boolean_With_Unknown*, and return a value of that type.

This support package also provides a conversion function, which converts the input of the type *Boolean_With_Unknown* to a value of the Ada predefined type boolean. If the input object has the value *Unknown*, then the *Null_Value_Error* exception is raised.

Finally, the package provides three testing functions that return boolean values. These functions, *Is_True*, *Is_False*, and *Is_Unknown*, return the value true if the input passes the test; otherwise the functions return the value false.

C.1.7 Operations Available to the Application

	Left	Operand Type Right	Result	Exceptions
All Domains				
Null_SQL_<type>_			_Type	
With_Null		_Not_Null	_Type	
Without_Null		_Type ¹	_Not_Null ²	Null_Value_Error
Is_Null ³ , Not_Null		_Type	Boolean	
Assign	_Type	_Type		Constraint_Error
Equals, Not_Equals	_Type	_Type	B_W_U ⁴	
<, >, <=, >=	_Type	_Type	B_W_U	
=, /=, >, <, >=, <=	_Type	_Type	Boolean	

Numeric Domains

unary +, -, Abs		_Type	_Type
+, -, /, *	_Type	_Type	_Type
**	_Type	Integer	_Type

Int and Smallint Domains

Mod, Rem	_Type	_Type	_Type
Image	_Type		SQL_Char
Image	_Not_Null		SQL_ChrNN ⁵
Value	SQL_Char		_Type
Value	SQL_ChrNN		_Not_Null

Character Domains

VERSION 2

Without_Null_Unpadded		_Type	_Not_Null	Null_Value_Error
To_String		_Not_Null	String	
To_String		_Type	String	Null_Value_Error
To_Unpadded_String		_Not_Null	String	
To_Unpadded_String		_Type	String	Null_Value_Error
To_SQL_Char_Not_Null		String	_Not_Null	
To_SQL_Char		String	_Type	
Unpadded_Length ¹⁰		_Type	SQL_U_L ⁹	Null_Value_Error
Substring ¹⁰		_Type	_Type	Constraint_Error
&	_Type	_Type	_Type	

Bit String Domains

Substring ¹⁰		_Type	_Type	Constraint_Error
&	_Type	_Type	_Type	

Enumeration Domains

Pred, Succ		_Type	_Type	
Image		_Type	SQL_Char	
Image		_Not_Null	SQL_Ch>NN	
Pos		_Type	Integer	Null_Value_Error
Val		Integer	_Type	
Value		SQL_Char	_Type	
Value		SQL_Ch>NN	_Not_Null	

Boolean Functions

not		B_W_U	Boolean	
and, or, xor	B_W_U	B_W_U	Boolean	
To_Boolean		B_W_U	Boolean	Null_Value_Error
Is_True,		B_W_U	Boolean	
Is_False,		B_W_U	Boolean	
Is_Unknown		B_W_U	Boolean	

1. "_Type" represents the type in the abstract domain of which objects that may be null are declared.
2. "_Not_Null" represents the type in the abstract domain of which objects that are not null may be declared.
3. "Assign" is a procedure. The result is returned in object "Left."
4. "B_W_U" is an abbreviation for Boolean_With_Unknown.
5. "SQL_Ch>NN" is an abbreviation for SQL_Char_Not_Null.
6. "SQL_Int>NN" is an abbreviation for SQL_Int_Not_Null.
7. "SQL_Db>NN" is an abbreviation for SQL_Double_Precision_Not_Null.
8. "SQL_DbI" is an abbreviation for SQL_Double_Precision.
9. "SQL_U_L" is an abbreviation for the SQL_Char_Pkg subtype SQL_Unpadded_Length.
10. Substring has two additional parameters Start and Length which are both of the SQL_Char_Pkg subtype SQL_Char_Length.

C.2 Standard Support Package Specifications

C.2.1 Interfaces.SQL

The package Interfaces.SQL is defined in 12.4 of ISO/IEC 9075:1992 and is reproduced here for information only.

```

package Interfaces.SQL is
  package Character_Set renames csp;
  subtype Character_Type is Character_Set.cst;
  type Char is array (positive range <>) of Character_Type;
  type Bit is array (natural range <>) of boolean;
  type Smallint is range bs..ts;
  type Int is range bi..ti;
  type Real is digits dr;
  type Double_Precision is digits dd;
  subtype Indicator_Type is t;
  type Sqlcode_Type is range bsc..tsc;
  subtype Sql_Error is Sqlcode_Type range Sqlcode_Type'FIRST .. -1;
  subtype Not_Found is Sqlcode_Type range 100..100;
  type SQLSTATE_Type is new Char (1 .. 5);
  -- csp is an implementor-defined package and cst is an
  -- implementor-defined character type. bs, ts, bi, ti, dr, dd, bsc,
  -- and tsc are implementor defined integral values. t is int or
  -- smallint corresponding to an implementor-defined exact
  -- numeric type of indicator parameters.
  package SQLSTATE_Codes is
    -- this subpackage contains a constant declaration for each
    -- SQLSTATE value defined by ISO/IEC 9075:1992. These constants
    --
    are reproduced in the SAMeDL module (and therefore the Ada package)
    -- SAMeDL_Standard. See Annex A of this specification.
    -- The list of constants declared in 12.4 of ISO/IEC 9075:1992
    -- is definitive.
  end SQLSTATE_Codes;
end Interfaces.SQL;

```

C.2.2 SQL_Boolean_Pkg

```

package SQL_Boolean_Pkg is
  type Boolean_with_Unknown is (FALSE, UNKNOWN, TRUE);
  ---- Three valued Logic operations ----
  --- three-val X three-val => three-val ---
  function "not" (Left : Boolean_with_Unknown)
    return Boolean_with_Unknown;
  pragma INLINE ("not");
  function "and" (Left, Right : Boolean_with_Unknown)
    return Boolean_with_Unknown;
  pragma INLINE ("and");
  function "or" (Left, Right : Boolean_with_Unknown)
    return Boolean_with_Unknown;
  pragma INLINE ("or");
  function "xor" (Left, Right : Boolean_with_Unknown)
    return Boolean_with_Unknown;
  pragma INLINE ("xor");
  --- three-val => bool or exception ---
  function To_Boolean (Left : Boolean_with_Unknown) return Boolean;
  pragma INLINE (To_Boolean);
  --- three-val => bool ---
  function Is_True (Left : Boolean_with_Unknown) return Boolean;
  pragma INLINE (Is_True);
  function Is_False (Left : Boolean_with_Unknown) return Boolean;
  pragma INLINE (Is_False);
  function Is_Unknown (Left : Boolean_with_Unknown) return Boolean;

```

```

    pragma INLINE (Is_Unknown);
end SQL_Boolean_Pkg;

```

C.2.3 SQL_Int_Pkg

```

with Interfaces.SQL;
with SQL_Boolean_Pkg; use SQL_Boolean_Pkg;
with SQL_Char_Pkg; use SQL_Char_Pkg;
package SQL_Int_Pkg is
    type SQL_Int_Not_Null is new Interfaces.SQL.Int;
    ---- Possibly Null Integer ----
    type SQL_Int is limited private;
    function Null_SQL_Int return SQL_Int;
    pragma INLINE (Null_SQL_Int);
    -- this pair of functions convert between the
    -- null-bearing and non-null-bearing types.
    function Without_Null_Base(Value : SQL_Int)
        return SQL_Int_Not_Null;
    pragma INLINE (Without_Null_Base);
    -- With_Null_Base raises Null_Value_Error if the input
    -- value is null
    function With_Null_Base(Value : SQL_Int_Not_Null)
        return SQL_Int;
    pragma INLINE (With_Null_Base);
    -- this procedure implements range checking
    -- note: it is not meant to be used directly
    -- by application programmers
    -- see the generic package SQL_Int_Ops
    -- raises constraint_error if not
    -- (First <= Right <= Last)
    procedure Assign_with_check (
        Left : in out SQL_Int; Right : SQL_Int;
        First, Last : SQL_Int_Not_Null);
    pragma INLINE (Assign_with_check);
    -- the following functions implement three valued
    -- arithmetic
    -- if either input to any of these functions is null
    -- the function returns the null value; otherwise
    -- they perform the indicated operation
    -- these functions raise no exceptions
    function "+"(Right : SQL_Int) return SQL_Int;
    pragma INLINE ("+");
    function "-"(Right : SQL_Int) return SQL_Int;
    pragma INLINE ("-");
    function "abs"(Right : SQL_Int) return SQL_Int;
    pragma INLINE ("abs");
    function "+"(Left, Right : SQL_Int) return SQL_Int;
    pragma INLINE ("+");
    function "*" (Left, Right : SQL_Int) return SQL_Int;
    pragma INLINE ("*");
    function "-"(Left, Right : SQL_Int) return SQL_Int;
    pragma INLINE ("-");
    function "/"(Left, Right : SQL_Int) return SQL_Int;
    pragma INLINE ("/");
    function "mod" (Left, Right : SQL_Int) return SQL_Int;
    pragma INLINE ("mod");
    function "rem" (Left, Right : SQL_Int) return SQL_Int;
    pragma INLINE ("rem");
    function "***" (Left : SQL_Int; Right: Integer) return SQL_Int;
    pragma INLINE ("***");
    -- simulation of 'IMAGE and 'VALUE that
    -- return/take SQL_Char[_Not_Null] instead of string
    function IMAGE (Left : SQL_Int_Not_Null) return SQL_Char_Not_Null;
    function IMAGE (Left : SQL_Int) return SQL_Char;
    pragma INLINE (IMAGE);
    function VALUE (Left : SQL_Char_Not_Null) return SQL_Int_Not_Null;
    function VALUE (Left : SQL_Char) return SQL_Int;

```

```

pragma INLINE (VALUE);
-- Logical Operations --
-- type X type => Boolean_with_unknown --
-- these functions implement three valued logic
-- if either input is the null value, the functions
-- return the truth value UNKNOWN; otherwise they
-- perform the indicated comparison.
-- these functions raise no exceptions
function Equals (Left, Right : SQL_Int)
    return Boolean_with_Unknown;
pragma INLINE (Equals);
function Not_Equals (Left, Right : SQL_Int)
    return Boolean_with_Unknown;
pragma INLINE (Not_Equals);
function "<" (Left, Right : SQL_Int) return Boolean_with_Unknown;
pragma INLINE ("<");
function ">" (Left, Right : SQL_Int) return Boolean_with_Unknown;
pragma INLINE (">");
function "<=" (Left, Right : SQL_Int) return Boolean_with_Unknown;
pragma INLINE ("<=");
function ">=" (Left, Right : SQL_Int) return Boolean_with_Unknown;
pragma INLINE (">=");
-- type => Boolean --
function Is_Null(Value : SQL_Int) return Boolean;
pragma INLINE (Is_Null);
function Not_Null(Value : SQL_Int) return Boolean;
pragma INLINE (Not_Null);
-- These functions of class type => Boolean
-- equate UNKNOWN with FALSE. That is, they return TRUE
-- only when the function returns TRUE. UNKNOWN and FALSE
-- are mapped to FALSE.
function "=" (Left, Right : SQL_Int) return Boolean;
pragma INLINE ("=");
function "<" (Left, Right : SQL_Int) return Boolean;
pragma INLINE ("<");
function ">" (Left, Right : SQL_Int) return Boolean;
pragma INLINE (">");
function "<=" (Left, Right : SQL_Int) return Boolean;
pragma INLINE ("<=");
function ">=" (Left, Right : SQL_Int) return Boolean;
pragma INLINE (">=");
-- this generic is instantiated once for every abstract
-- domain based on the SQL type Int.
-- the three subprogram formal parameters are meant to
-- default to the programs declared above.
-- that is, the package should be instantiated in the
-- scope of a use clause for SQL_Int_Pkg.
-- the two actual types together form the abstract
-- domain.
-- the purpose of the generic is to create functions
-- which convert between the two actual types and a
-- procedure which implements a range constrained
-- assignment for the null-bearing type.
-- the bodies of these subprograms are calls to
-- subprograms declared above and passed as defaults to
-- the generic.
generic
    type With_Null_type is limited private;
    type Without_null_type is range <>;
    with function With_Null_Base(Value : SQL_Int_Not_Null)
        return With_Null_Type is <>;
    with function Without_Null_Base(Value : With_Null_Type)
        return SQL_Int_Not_Null is <>;
    with procedure Assign_with_check (
        Left : in out With_Null_Type; Right : With_Null_Type;
        First, Last : SQL_Int_Not_Null) is <>;
package SQL_Int_Ops is
    function With_Null (Value : Without_Null_type)

```

```

        return With_Null_type;
    pragma INLINE (With_Null);
    function Without_Null (Value : With_Null_Type)
        return Without_Null_type;
    pragma INLINE (Without_Null);
    procedure assign (
        Left  : in out With_null_Type;
        Right : in With_null_type);
    pragma INLINE (assign);
end SQL_Int_Ops;
private
type SQL_Int is record
    Is_Null: Boolean := True;
    Value: SQL_Int_Not_Null;
end record;
end SQL_Int_Pkg;

```

C.2.4 SQL_Smallint_Pkg

```

with Interfaces.SQL;
with SQL_Boolean_Pkg; use SQL_Boolean_Pkg;
with SQL_Char_Pkg; use SQL_Char_Pkg;
package SQL_Smallint_Pkg is
    type SQL_Smallint_Not_Null is new Interfaces.SQL.Smallint;
    ---- Possibly Null Integer ----
    type SQL_Smallint is limited private;
    function Null_SQL_Smallint return SQL_Smallint;
    pragma INLINE (Null_SQL_Smallint);
    -- this pair of functions converts between the
    -- null-bearing and not null-bearing types.
    function Without_Null_Base(Value : SQL_Smallint)
        return SQL_Smallint_Not_Null;
    pragma INLINE (Without_Null_Base);
    -- With_Null_Base raises Null_Value_Error if the input
    -- value is null
    function With_Null_Base(Value : SQL_Smallint_Not_Null)
        return SQL_Smallint;
    pragma INLINE (With_Null_Base);
    -- this procedure implements range checking
    -- note: it is not meant to be used directly
    -- by application programmers
    -- see the generic package SQL_Smallint_Op
    -- raises constraint_error if not
    -- (First <= Right <= Last)
    procedure Assign_with_check (
        Left : in out SQL_Smallint; Right : SQL_Smallint;
        First, Last : SQL_Smallint_Not_Null);
    pragma INLINE (Assign_with_check);
    -- the following functions implement three valued
    -- arithmetic
    -- if either input to any of these functions is null
    -- the function returns the null value; otherwise
    -- they perform the indicated operation
    -- these functions raise no exceptions
    function "+"(Right : SQL_Smallint) return SQL_Smallint;
    pragma INLINE ("+" );
    function "--"(Right : SQL_Smallint) return SQL_Smallint;
    pragma INLINE ("--" );
    function "abs"(Right : SQL_Smallint) return SQL_Smallint;
    pragma INLINE ("abs" );
    function "+"(Left, Right : SQL_Smallint) return SQL_Smallint;
    pragma INLINE ("+" );
    function "*" (Left, Right : SQL_Smallint) return SQL_Smallint;
    pragma INLINE ("*" );
    function "--"(Left, Right : SQL_Smallint) return SQL_Smallint;
    pragma INLINE ("--" );
    function "/"(Left, Right : SQL_Smallint) return SQL_Smallint;

```

```

pragma INLINE ("/");
function "mod" (Left, Right : SQL_Smallint) return SQL_Smallint;
pragma INLINE ("mod");
function "rem" (Left, Right : SQL_Smallint) return SQL_Smallint;
pragma INLINE ("rem");
function "***" (Left : SQL_Smallint; Right: Integer)
    return SQL_Smallint;
pragma INLINE ("***");
-- simulation of 'IMAGE and 'VALUE that
-- return/take SQL_Char[_Not_Null] instead of string
function IMAGE (Left : SQL_Smallint_Not_Null)
    return SQL_Char_Not_Null;
function IMAGE (Left : SQL_Smallint) return SQL_Char;
pragma INLINE (IMAGE);
function VALUE (Left : SQL_Char_Not_Null)
    return SQL_Smallint_Not_Null;
function VALUE (Left : SQL_Char) return SQL_Smallint;
pragma INLINE (VALUE);
-- Logical Operations --
-- type X type => Boolean_with_unknown --
-- these functions implement three valued logic
-- if either input is the null value, the functions
-- return the truth value UNKNOWN; otherwise they
-- perform the indicated comparison.
-- these functions raise no exceptions
function Equals (Left, Right : SQL_Smallint)
    return Boolean_with_Unknown;
pragma INLINE (Equals);
function Not_Equals (Left, Right : SQL_Smallint)
    return Boolean_with_Unknown;
pragma INLINE (Not_Equals);
function "<" (Left, Right : SQL_Smallint)
    return Boolean_with_Unknown;
pragma INLINE ("<");
function ">" (Left, Right : SQL_Smallint)
    return Boolean_with_Unknown;
pragma INLINE (">");
function "<=" (Left, Right : SQL_Smallint)
    return Boolean_with_Unknown;
pragma INLINE ("<=");
function ">=" (Left, Right : SQL_Smallint)
    return Boolean_with_Unknown;
pragma INLINE (">=");
-- type => Boolean --
function Is_Null(Value : SQL_Smallint) return Boolean;
pragma INLINE (Is_Null);
function Not_Null(Value : SQL_Smallint) return Boolean;
pragma INLINE (Not_Null);
-- These functions of class type => Boolean
-- equate UNKNOWN with FALSE. That is, they return TRUE
-- only when the function returns TRUE. UNKNOWN and FALSE
-- are mapped to FALSE.
function "=" (Left, Right : SQL_Smallint) return Boolean;
pragma INLINE ("=");
function "<" (Left, Right : SQL_Smallint) return Boolean;
pragma INLINE ("<");
function ">" (Left, Right : SQL_Smallint) return Boolean;
pragma INLINE (">");
function "<=" (Left, Right : SQL_Smallint) return Boolean;
pragma INLINE ("<=");
function ">=" (Left, Right : SQL_Smallint) return Boolean;
pragma INLINE (">=");
-- this generic is instantiated once for every abstract
-- domain based on the SQL type Smallint.
-- the three subprogram formal parameters are meant to
-- default to the programs declared above.
-- that is, the package should be instantiated in the
-- scope of a use clause for SQL_Smallint_Pkg.

```

VERSION 2

```
-- the two actual types together form the abstract
-- domain.
-- the purpose of the generic is to create functions
-- which convert between the two actual types and a
-- procedure which implements a range constrained
-- assignment for the null-bearing type.
-- the bodies of these subprograms are calls to
-- subprograms declared above and passed as defaults to
-- the generic.
generic
  type With_Null_type is limited private;
  type Without_null_type is range <>;
  with function With_Null_Base(Value : SQL_Smallint_Not_Null)
    return With_Null_Type is <>;
  with function Without_Null_Base(Value : With_Null_Type)
    return SQL_Smallint_Not_Null is <>;
  with procedure Assign_with_check (
    Left : in out With_Null_Type; Right : With_Null_Type;
    First, Last : SQL_Smallint_Not_Null) is <>;
package SQL_Smallint_Ops is
  function With_Null (Value : Without_Null_type)
    return With_Null_type;
  pragma INLINE (With_Null);
  function Without_Null (Value : With_Null_Type)
    return Without_Null_type;
  pragma INLINE (Without_Null);
  procedure assign (
    Left : in out With_null_Type;
    Right : in With_null_type);
  pragma INLINE (assign);
end SQL_Smallint_Ops;
private
  type SQL_Smallint is record
    Is_Null: Boolean := True;
    Value: SQL_Smallint_Not_Null;
  end record;
end SQL_Smallint_Pkg;
```

C.2.5 SQL_Real_Pkg

```
with Interfaces.SQL;
with SQL_Boolean_Pkg; use SQL_Boolean_Pkg;
package SQL_Real_Pkg is
  type SQL_Real_Not_Null is new Interfaces.SQL.Real;
  ---- Possibly Null Real ----
  type SQL_Real is limited private;
  function Null_SQL_Real return SQL_Real;
  pragma INLINE (Null_SQL_Real);
  -- this pair of functions converts between the
  -- null-bearing and not null-bearing types
  function Without_Null_Base(Value : SQL_Real)
    return SQL_Real_Not_Null;
  pragma INLINE (Without_Null_Base);
  -- With_Null_Base raises Null_Value_Error if the input
  -- value is null
  function With_Null_Base(Value : SQL_Real_Not_Null)
    return SQL_Real;
  pragma INLINE (With_Null_Base);
  -- this procedure implements range checking
  -- note: it is not meant to be used directly
  -- by application programmers
  -- see the generic package SQL_Real_Ops
  -- raises constraint_error if not
  -- (First <= Right <= Last)
  procedure Assign_with_Check (
    Left : in out SQL_Real; Right : SQL_Real;
    First, Last : SQL_Real_Not_Null);
```

```

pragma INLINE (Assign_with_Check);
-- the following functions implement three valued
-- arithmetic
-- if either input to any of these functions is null
-- the function returns the null value; otherwise
-- they perform the indicated operation
-- these functions raise no exceptions
function "+"(Right : SQL_Real) return SQL_Real;
pragma INLINE ("+");
function "-"(Right : SQL_Real) return SQL_Real;
pragma INLINE ("-");
function "abs"(Right : SQL_Real) return SQL_Real;
pragma INLINE ("abs");
function "+"(Left, Right : SQL_Real) return SQL_Real;
pragma INLINE ("+");
function "*" (Left, Right : SQL_Real) return SQL_Real;
pragma INLINE ("*");
function "-"(Left, Right : SQL_Real) return SQL_Real;
pragma INLINE ("-");
function "/"(Left, Right : SQL_Real) return SQL_Real;
pragma INLINE ("/");
function "***"(Left : SQL_Real; Right : Integer) return SQL_Real;
pragma INLINE ("***");
-- Logical Operations --
-- type X type => Boolean_with_unknown --
-- these functions implement three valued logic
-- if either input is the null value, the functions
-- return the truth value UNKNOWN; otherwise they
-- perform the indicated comparison.
-- these functions raise no exceptions
function Equals (Left, Right : SQL_Real)
return Boolean_with_Unknown;
pragma INLINE (Equals);
function Not_Equals (Left, Right : SQL_Real)
return Boolean_with_Unknown;
pragma INLINE (Not_Equals);
function "<" (Left, Right : SQL_Real) return Boolean_with_Unknown;
pragma INLINE ("<");
function ">" (Left, Right : SQL_Real) return Boolean_with_Unknown;
pragma INLINE (">");
function "<=" (Left, Right : SQL_Real) return Boolean_with_Unknown;
pragma INLINE ("<=");
function ">=" (Left, Right : SQL_Real) return Boolean_with_Unknown;
pragma INLINE (">=");
-- type => Boolean --
function Is_Null(Value : SQL_Real) return Boolean;
pragma INLINE (Is_Null);
function Not_Null(Value : SQL_Real) return Boolean;
pragma INLINE (Not_Null);
-- These functions of class type => Boolean
-- equate UNKNOWN with FALSE. That is, they return TRUE
-- only when the function returns TRUE. UNKNOWN and FALSE
-- are mapped to FALSE.
function "=" (Left, Right : SQL_Real) return Boolean;
pragma INLINE ("=");
function "<" (Left, Right : SQL_Real) return Boolean;
pragma INLINE ("<");
function ">" (Left, Right : SQL_Real) return Boolean;
pragma INLINE (">");
function "<=" (Left, Right : SQL_Real) return Boolean;
pragma INLINE ("<=");
function ">=" (Left, Right : SQL_Real) return Boolean;
pragma INLINE (">=");
-- this generic is instantiated once for every abstract
-- domain based on the SQL type Real.
-- the three subprogram formal parameters are meant to
-- default to the programs declared above.
-- that is, the package should be instantiated in the

```

VERSION 2

```
-- scope of the use clause for SQL_Real_Pkg.
-- the two actual types together form the abstract
-- domain.
-- the purpose of the generic is to create functions
-- which convert between the two actual types and a
-- procedure which implements a range constrained
-- assignment for the null-bearing type.
-- the bodies of these subprograms are calls to
-- subprograms declared above and passed as defaults to
-- the generic.
generic
  type With_Null_type is limited private;
  type Without_null_type is digits <>;
  with function With_Null_Base(Value : SQL_Real_Not_Null)
    return With_Null_Type is <>;
  with function Without_Null_Base(Value : With_Null_Type)
    return SQL_Real_Not_Null is <>;
  with procedure Assign_with_check (
    Left : in out With_Null_Type; Right : With_Null_Type;
    First, Last : SQL_Real_Not_Null) is <>;
package SQL_Real_Ops is
  function With_Null (Value : Without_Null_type)
    return With_Null_type;
  pragma INLINE (With_Null);
  function Without_Null (Value : With_Null_Type)
    return Without_Null_type;
  pragma INLINE (Without_Null);
  procedure assign (
    Left : in out With_Null_Type;
    Right : in With_Null_type);
  pragma INLINE (assign);
end SQL_Real_Ops;
private
  type SQL_Real is record
    Is_Null: Boolean := True;
    Value: SQL_Real_Not_Null;
  end record;
end SQL_Real_Pkg;
```

C.2.6 SQL_Double_Precision_Pkg

```
with Interfaces.SQL;
with SQL_Boolean_Pkg; use SQL_Boolean_Pkg;
package SQL_Double_Precision_Pkg is
  type SQL_Double_Precision_Not_Null is new
    Interfaces.SQL.Double_Precision;
  ---- Possibly Null Double_Precision ----
  type SQL_Double_Precision is limited private;
  function Null_SQL_Double_Precision return SQL_Double_Precision;
  pragma INLINE (Null_SQL_Double_Precision);
  -- this pair of functions converts between the
  -- null-bearing and not null-bearing types.
  function Without_Null_Base(Value : SQL_Double_Precision)
    return SQL_Double_Precision_Not_Null;
  pragma INLINE (Without_Null_Base);
  -- With_Null_Base raises Null_Value_Error if the input
  -- value is null
  function With_Null_Base(Value : SQL_Double_Precision_Not_Null)
    return SQL_Double_Precision;
  pragma INLINE (With_Null_Base);
  -- this procedure implements range checking
  -- note: it is not meant to be used directly
  -- by application programmers
  -- see the generic package SQL_Double_Precision_Op
  -- raises constraint_error if not
  -- (First <= Right <= Last)
  procedure Assign_with_Check (
```

```

    Left : in out SQL_Double_Precision;
    Right : SQL_Double_Precision;
    First, Last : SQL_Double_Precision_Not_Null);
pragma INLINE (Assign_with_Check);
-- the following functions implement three valued
-- arithmetic
-- if either input to any of these functions is null
-- the function returns the null value; otherwise they
-- perform the indicated operation
-- these functions raise no exceptions
function "+"(Right : SQL_Double_Precision)
    return SQL_Double_Precision;
pragma INLINE ("+");
function "-"(Right : SQL_Double_Precision)
    return SQL_Double_Precision;
pragma INLINE ("-");
function "abs"(Right : SQL_Double_Precision)
    return SQL_Double_Precision;
pragma INLINE ("abs");
function "+"(Left, Right : SQL_Double_Precision)
    return SQL_Double_Precision;
pragma INLINE ("+");
function "*" (Left, Right : SQL_Double_Precision)
    return SQL_Double_Precision;
pragma INLINE ("*");
function "-"(Left, Right : SQL_Double_Precision)
    return SQL_Double_Precision;
pragma INLINE ("-");
function "/"(Left, Right : SQL_Double_Precision)
    return SQL_Double_Precision;
pragma INLINE ("/");
function "***"(Left : SQL_Double_Precision; Right : Integer)
    return SQL_Double_Precision;
pragma INLINE ("***");
-- Logical Operations --
-- type X type => Boolean_with_unknown --
-- these functions implement three valued logic
-- if either input is the null value, the functions
-- return the truth value UNKNOWN; otherwise they
-- perform the indicated comparison.
-- these functions raise no exceptions
function Equals (Left, Right : SQL_Double_Precision)
    return Boolean_with_Unknown;
pragma INLINE (Equals);
function Not_Equals (Left, Right : SQL_Double_Precision)
    return Boolean_with_Unknown;
pragma INLINE (Not_Equals);
function "<" (Left, Right : SQL_Double_Precision)
    return Boolean_with_Unknown;
pragma INLINE ("<");
function ">" (Left, Right : SQL_Double_Precision)
    return Boolean_with_Unknown;
pragma INLINE (">");
function "<=" (Left, Right : SQL_Double_Precision)
    return Boolean_with_Unknown;
pragma INLINE ("<=");
function ">=" (Left, Right : SQL_Double_Precision)
    return Boolean_with_Unknown;
pragma INLINE (">=");
-- type => Boolean --
function Is_Null(Value : SQL_Double_Precision) return Boolean;
pragma INLINE (Is_Null);
function Not_Null(Value : SQL_Double_Precision) return Boolean;
pragma INLINE (Not_Null);
-- These functions of class type => Boolean
-- equate UNKNOWN with FALSE. That is, they return TRUE
-- only when the function returns TRUE. UNKNOWN and FALSE
-- are mapped to FALSE.

```

```

function "=" (Left, Right : SQL_Double_Precision) return Boolean;
function "<" (Left, Right : SQL_Double_Precision) return Boolean;
function ">" (Left, Right : SQL_Double_Precision) return Boolean;
function "<=" (Left, Right : SQL_Double_Precision) return Boolean;
function ">=" (Left, Right : SQL_Double_Precision) return Boolean;
-- this generic is instantiated once for every abstract
-- domain based on the SQL type Double_Precision.
-- the three subprogram formal parameters are meant to
-- default to the programs declared above.
-- that is, the package should be instantiated in the
-- scope of the use clause for
--   SQL_Double_Precision_Pkg.
-- the two actual types together form the abstract
-- domain.
-- the purpose of the generic is to create functions
-- which convert between the two actual types and a
-- procedure which implements a range constrained
-- assignment for the null-bearing type.
-- the bodies of these subprograms are calls to
-- subprograms declared above and passed as defaults
-- to the generic.
generic
  type With_Null_type is limited private;
  type Without_null_type is digits <>;
  with function With_Null_Base(
    Value : SQL_Double_Precision_Not_Null)
    return With_Null_Type is <>;
  with function Without_Null_Base(Value : With_Null_Type)
    return SQL_Double_Precision_Not_Null is <>;
  with procedure Assign_with_check (
    Left : in out With_Null_Type; Right : With_Null_Type;
    First, Last : SQL_Double_Precision_Not_Null) is <>;
package SQL_Double_Precision_Ops is
  function With_Null (Value : Without_Null_type)
    return With_Null_type;
  pragma INLINE (With_Null);
  function Without_Null (Value : With_Null_Type)
    return Without_Null_type;
  pragma INLINE (Without_Null);
  procedure assign (
    Left : in out With_null_Type;
    Right : in With_null_type);
  pragma INLINE (assign);
end SQL_Double_Precision_Ops;
private
  type SQL_Double_Precision is record
    Is_Null: Boolean := True;
    Value: SQL_Double_Precision_Not_Null;
  end record;
end SQL_Double_Precision_Pkg;

```

C.2.7 SQL_Char_Pkg

```

with SQL_System; use SQL_System;
with SQL_Boolean_Pkg; use SQL_Boolean_Pkg;
with Interfaces.SQL;
package SQL_Char_Pkg is
  subtype SQL_Char_Length is natural
    range 1 .. MAXCHRLEN;
  subtype SQL_Unpadded_Length is natural
    range 0 .. MAXCHRLEN;
  type SQL_Char_Not_Null is new Interfaces.SQL.Char;
  type SQL_Char(Length : SQL_Char_Length) is limited private;
  function Null_SQL_Char return SQL_Char;
  pragma INLINE (Null_SQL_Char);
  -- the next three functions convert between
  -- null-bearing and not null-bearing-types

```

```

-- Without_Null_Base and With_Null_Base are
-- inverses (mod. null values)
-- see also SQL_Char_Ops generic package below
function With_Null_Base(Value : SQL_Char_Not_Null)
    return SQL_Char;
pragma INLINE (With_Null_Base);
-- Without_Null_Base and Without_Null_Base_Unpadded raise
-- null_value_error on the null input
function Without_Null_Base(Value : SQL_Char)
    return SQL_Char_Not_Null;
pragma INLINE (Without_Null_Base);
-- Without_Null_Unpadded_Base removes trailing blanks from
-- the input
function Without_Null_Unpadded_Base(Value : SQL_Char)
    return SQL_Char_Not_Null;
pragma INLINE (Without_Null_Unpadded_Base);
-- axiom: unpadded_Length(x) =
-- Without_Null_Unpadded_Base(x)'Length
-- both functions raise null_value_error if x is null
-- the next six functions convert between Standard.String
-- types and the SQL_Char and SQL_Char_Not_Null types
function To_String (Value : SQL_Char_Not_Null)
    return String;
function To_String (Value : SQL_Char)
    return String;
function To_Unpadded_String (Value : SQL_Char_Not_Null)
    return String;
function To_Unpadded_String (Value : SQL_Char)
    return String;
pragma INLINE (To_Unpadded_String);
-- this INLINE works for BOTH functions!!
function To_SQL_Char_Not_Null (Value : String)
    return SQL_Char_Not_Null;
function To_SQL_Char (Value : String)
    return SQL_Char;
pragma INLINE (To_SQL_Char);
function Unpadded_Length (Value : SQL_Char)
    return SQL_Unpadded_Length;
pragma INLINE (Unpadded_Length);
procedure Assign(
    Left : out SQL_Char;
    Right : SQL_Char);
pragma INLINE (Assign);
-- Substring(x,k,m) returns the substring of x starting
-- at position k (relative to 1) with length m.
-- returns null value if x is null
-- raises constraint_error if Start < 1 or Length < 1 or
-- Start + Length - 1 > x.Length
function Substring (
    Value : SQL_Char;
    Start, Length : SQL_Char_Length)
    return SQL_Char;
pragma INLINE (Substring);
-- "&" returns null if either parameter is null;
-- otherwise performs concatenation in the usual way,
-- preserving all blanks.
-- may raise constraint_error implicitly if result is
-- too large (i.e., greater than SQL_Char_Length'Last
function "&" (Left, Right : SQL_Char)
    return SQL_Char;
pragma INLINE ("&");
-- Logical Operations --
-- type X type => Boolean_with_unknown --
-- the comparison operators return the boolean value
-- UNKNOWN if either parameter is null; otherwise,
-- the comparison is done in accordance with
-- ANSI X3.135-1986 para 5.11 general rule 5; that is,
-- the shorter of the two string parameters is

```

VERSION 2

```

-- effectively padded with blanks to be the length of
-- the longer string and a standard Ada comparison is
-- then made
function Equals (Left, Right : SQL_Char)
  return Boolean_with_Unknown;
pragma INLINE (Equals);
function Not_Equals (Left, Right : SQL_Char)
  return Boolean_with_Unknown;
pragma INLINE (Not_Equals);
function "<" (Left, Right : SQL_Char) return Boolean_with_Unknown;
pragma INLINE ("<");
function ">" (Left, Right : SQL_Char) return Boolean_with_Unknown;
pragma INLINE (">");
function "<=" (Left, Right : SQL_Char) return Boolean_with_Unknown;
pragma INLINE ("<=");
function ">=" (Left, Right : SQL_Char) return Boolean_with_Unknown;
pragma INLINE (">=");
-- type => Boolean --
function Is_Null (Value : SQL_Char) return Boolean;
pragma INLINE (Is_Null);
function Not_Null (Value : SQL_Char) return Boolean;
pragma INLINE (Not_Null);
-- These functions of class type => Boolean
-- equate UNKNOWN with FALSE. That is, they return TRUE
-- only when the function returns TRUE. UNKNOWN and FALSE
-- are mapped to FALSE.
function "=" (Left, Right : SQL_Char) return Boolean;
pragma INLINE ("=");
function "<" (Left, Right : SQL_Char) return Boolean;
pragma INLINE ("<");
function ">" (Left, Right : SQL_Char) return Boolean;
pragma INLINE (">");
function "<=" (Left, Right : SQL_Char) return Boolean;
pragma INLINE ("<=");
function ">=" (Left, Right : SQL_Char) return Boolean;
pragma INLINE (">=");
-- the purpose of the following generic is to generate
-- conversion functions between a type derived from
-- SQL_Char_Not_Null, which are effectively Ada
-- strings and a type derived from SQL_Char, which
-- mimic the behavior of SQL strings.
-- the subprogram formal is meant to default; that is,
-- this generic should be instantiated in the scope
-- of an use clause for SQL_Char_Pkg.
generic
  type With_Null_Type is limited private;
  type Without_Null_Type is array (positive range <>)
    of Interfaces.SQL.Character_Type;
  with function With_Null_Base (Value : SQL_Char_Not_Null)
    return With_Null_Type is <>;
  with function Without_Null_Base (Value : With_Null_Type)
    return SQL_Char_Not_Null is <>;
  with function Without_Null_Unpadded_Base (Value : With_Null_Type)
    return SQL_Char_Not_Null is <>;
package SQL_Char_Ops is
  function With_Null (Value : Without_Null_Type)
    return With_Null_Type;
  pragma INLINE (With_Null);
  function Without_Null (Value : With_Null_Type)
    return Without_Null_Type;
  pragma INLINE (Without_Null);
  function Without_Null_Unpadded (Value : With_Null_Type)
    return Without_Null_Type;
  pragma INLINE (Without_Null_Unpadded);
end SQL_Char_Ops;
private
  type SQL_Char (Length : SQL_Char_Length) is record
    Is_Null : Boolean := True;

```

```

        Unpadded_Length: SQL_Unpadded_Length;
        Text: SQL_Char_Not_Null(1 .. Length);
    end record;
end SQL_Char_Pkg;

```

C.2.8 SQL_VarChar_Pkg

```

-- mimics Ada.Strings.Bounded while also adding 3-valued
-- comparison operators and conversion functions
with Ada.Strings.Maps;
with Ada.Finalization;
with Interfaces.SQL, Interfaces.SQL.Varying;
package SQL_VarChar is
    subtype SQL_VarChar_Not_Null is Interfaces.SQL.Varying.Char;
    type SQL_VarChar is private;
    function Null_SQL_VarChar return SQL_VarChar;
    function
    function Length (Source : SQL_VarChar) return Natural;
    function To_SQL_VarChar (Source : Interfaces.SQL.Char)
        return SQL_VarChar;
    -- Without_Null_Base and With_Null_Base are
    -- inverses (mod. null values)
    -- see also SQL_Char_Ops generic package below
    function With_Null_Base(Value : SQL_VarChar_Not_Null)
        return SQL_Char;
    -- Without_Null_Base and Without_Null_Base_Unpadded raise
    -- null_value_error on the null input
    function Without_Null_Base(Value : SQL_VarChar)
        return SQL_VarChar_Not_Null;
    function To_SQL_VarChar (Length : in Natural) return SQL_VarChar;
    function To_SQL_Char (Source : SQL_VarChar) return Interfaces.SQL.Char;
    procedure Append (Source : in out SQL_VarChar;
        New_Item : in SQL_VarChar);
    procedure Append (Source : in out SQL_VarChar;
        New_Item : in Interfaces.SQL.Char);
    procedure Append (Source : in out SQL_VarChar;
        New_Item : in Character);
    function "&" (Left, Right : SQL_VarChar) return SQL_VarChar;
    function "&" (Left : in SQL_VarChar;
        Right : in Interfaces.SQL.Char) return SQL_VarChar;
    function "&" (Left : in Interfaces.SQL.Char;
        Right : in SQL_VarChar) return SQL_VarChar;
    function "&" (Left : in SQL_VarChar;
        Right : in Character) return SQL_VarChar;
    function "&" (Left : in Character;
        Right : in SQL_VarChar) return SQL_VarChar;
    function Element (Source : in SQL_VarChar;
        Index : in Positive) return Character;
    procedure Replace_Element (Source : in out SQL_VarChar;
        Index : in Positive; By : Character);
    function Slice (Source : in SQL_VarChar;
        Low : in Positive; High : in Natural)
        return Interfaces.SQL.Char;
    function "=" (Left, Right : in SQL_VarChar)
        return Boolean_with_Unknown;
    function "=" (Left : in SQL_VarChar;
        Right : in Interfaces.SQL.Char) return Boolean_with_Unknown;
    function "=" (Left : in Interfaces.SQL.Char;
        Right : in SQL_VarChar) return Boolean_with_Unknown;
    function "<" (Left, Right : in SQL_VarChar)
        return Boolean_with_Unknown;
    function "<" (Left : in SQL_VarChar;
        Right : in Interfaces.SQL.Char) return Boolean_with_Unknown;
    function "<" (Left : in Interfaces.SQL.Char;
        Right : in SQL_VarChar) return Boolean_with_Unknown;
    function "<=" (Left, Right : in SQL_VarChar)
        return Boolean_with_Unknown;

```

```

function "<=" (Left : in SQL_VarChar;
  Right : in Interfaces.SQL.Char) return Boolean_with_Unknown;
function "<=" (Left : in Interfaces.SQL.Char;
  Right : in SQL_VarChar) return Boolean_with_Unknown;
function ">" (Left, Right : in SQL_VarChar)
  return Boolean_with_Unknown;
function ">" (Left : in SQL_VarChar;
  Right : in Interfaces.SQL.Char) return Boolean_with_Unknown;
function ">" (Left : in Interfaces.SQL.Char;
  Right : in SQL_VarChar) return Boolean_with_Unknown;
function ">=" (Left, Right : in SQL_VarChar)
  return Boolean_with_Unknown;
function ">=" (Left : in SQL_VarChar;
  Right : in Interfaces.SQL.Char) return Boolean_with_Unknown;
function ">=" (Left : in Interfaces.SQL.Char;
  Right : in SQL_VarChar) return Boolean_with_Unknown;
-- these versions of the relationship tests
-- equate UNKNOWN with FALSE
function "=" (Left, Right : in SQL_VarChar) return Boolean;
function "=" (Left : in SQL_VarChar;
  Right : in Interfaces.SQL.Char) return Boolean;
function "=" (Left : in Interfaces.SQL.Char;
  Right : in SQL_VarChar) return Boolean;
function "<" (Left, Right : in SQL_VarChar) return Boolean;
function "<" (Left : in SQL_VarChar;
  Right : in Interfaces.SQL.Char) return Boolean;
function "<" (Left : in Interfaces.SQL.Char;
  Right : in SQL_VarChar) return Boolean;
function "<=" (Left, Right : in SQL_VarChar) return Boolean;
function "<=" (Left : in SQL_VarChar;
  Right : in Interfaces.SQL.Char) return Boolean;
function "<=" (Left : in Interfaces.SQL.Char;
  Right : in SQL_VarChar) return Boolean;
function ">" (Left, Right : in SQL_VarChar) return Boolean;
function ">" (Left : in SQL_VarChar;
  Right : in Interfaces.SQL.Char) return Boolean;
function ">" (Left : in Interfaces.SQL.Char;
  Right : in SQL_VarChar) return Boolean;
function ">=" (Left, Right : in SQL_VarChar) return Boolean;
function ">=" (Left : in SQL_VarChar;
  Right : in Interfaces.SQL.Char) return Boolean;
function ">=" (Left : in Interfaces.SQL.Char;
  Right : in SQL_VarChar) return Boolean;
----- -- Search Subprograms -- -----
function Index (Source : in SQL_VarChar;
  Pattern : in Interfaces.SQL.Char;
  Going : in Direction := Forward;
  Mapping : in Maps.Character_Mapping := Maps.Identity)
  return Natural;
function Index (Source : in SQL_VarChar;
  Pattern : in Interfaces.SQL.Char;
  Going : in Direction := Forward;
  Mapping : in Maps.Character_Mapping_Function)
  return Natural;
function Index (Source : in SQL_VarChar;
  Set : in Maps.Character_Set;
  Test : in Membership := Inside;
  Going : in Direction := Forward)
  return Natural;
function Index_Non_Blank (Source : in SQL_VarChar;
  Going : in Direction := Forward)
  return Natural;
function Count (Source : in SQL_VarChar;
  Pattern : in Interfaces.SQL.Char;
  Mapping : in Maps.Character_Mapping := Maps.Identity)
  return Natural;
function Count (Source : in SQL_VarChar;
  Pattern : in Interfaces.SQL.Char;

```

```

        Mapping : in Maps.Character_Mapping_Function)
    return Natural;
function Count (Source : in SQL_VarChar;
    Set : in Maps.Character_Set)
    return Natural;
procedure Find_Token (Source : in SQL_VarChar;
    Set : in Maps.Character_Set;
    Test : in Membership;
    First : out Positive; Last : out Natural);
function Translate (Source : in SQL_VarChar;
    Mapping : in Maps.Character_Mapping)
    return SQL_VarChar;
procedure Translate (Source : in out SQL_VarChar;
    Mapping : Maps.Character_Mapping);
function Translate (Source : in SQL_VarChar;
    Mapping : in Maps.Character_Mapping_Function)
    return SQL_VarChar;
procedure Translate (Source : in out SQL_VarChar;
    Mapping : in Maps.Character_Mapping_Function);
function Replace_Slice (Source : in SQL_VarChar;
    Low : in Positive;
    High : in Natural;
    By : in Interfaces.SQL.Char)
    return SQL_VarChar;
procedure Replace_Slice (Source : in out SQL_VarChar;
    Low : in Positive;
    High : in Natural;
    By : in Interfaces.SQL.Char);
function Insert (Source : in SQL_VarChar;
    Before : in Positive;
    New_Item : in Interfaces.SQL.Char)
    return SQL_VarChar;
procedure Insert (Source : in out SQL_VarChar;
    Before : in Positive;
    New_Item : in Interfaces.SQL.Char);
function Overwrite (Source : in SQL_VarChar;
    Position : in Positive;
    New_Item : in Interfaces.SQL.Char)
    return SQL_VarChar;
procedure Overwrite (Source : in out SQL_VarChar;
    Position : in Positive;
    New_Item : in Interfaces.SQL.Char);
function Delete (Source : in SQL_VarChar;
    From : in Positive;
    Through : in Natural)
    return SQL_VarChar;
procedure Delete (Source : in out SQL_VarChar;
    From : in Positive;
    Through : in Natural);
function Trim (Source : in SQL_VarChar;
    Side : in Trim_End)
    return SQL_VarChar;
procedure Trim (Source : in out SQL_VarChar;
    Side : in Trim_End);
function Trim (Source : in SQL_VarChar;
    Left : in Maps.Character_Set;
    Right : in Maps.Character_Set)
    return SQL_VarChar;
procedure Trim (Source : in out SQL_VarChar;
    Left : in Maps.Character_Set;
    Right : in Maps.Character_Set);
function Head (Source : in SQL_VarChar;
    Count : in Natural;
    Pad : in Character := Space)
    return SQL_VarChar;
procedure Head (Source : in out SQL_VarChar;
    Count : in Natural;
    Pad : in Character := Space);

```

VERSION 2

```
function Tail (Source : in SQL_VarChar;
  Count : in Natural;
  Pad : in Character := Space)
  return SQL_VarChar;
procedure Tail (Source : in out SQL_VarChar;
  Count : in Natural;
  Pad : in Character := Space);
function "*" (Left : in Natural;
  Right : in Character)
  return SQL_VarChar;
function "*" (Left : in Natural;
  Right : in Interfaces.SQL.Char)
  return SQL_VarChar;
function "*" (Left : in Natural;
  Right : in SQL_VarChar)
  return SQL_VarChar;
generic
  type With_Null_Type is limited private;
  type Without_Null_Type is limited private
  with function With_Null_Base (Value: Without_Null_Type)
    return With_Null_Type is <>;
  with function Without_Null_Base (Value: With_Null_Type)
    return SQL_VarChar_Not_Null is <>;
  with function Without_Null_Unpadded_Base (Value: With_Null_Type)
    return SQL_VarChar_Not_Null is <>;
  package SQL_VarChar_Ops is
    function With_Null (Value : Without_Null_Type)
      return With_Null_Type;
    pragma INLINE (With_Null);
    function Without_Null (Value : With_Null_Type)
      return Without_Null_Type;
    pragma INLINE (Without_Null);
    function Without_Null_Unpadded (Value : With_Null_Type)
      return Without_Null_Type;
    pragma INLINE (Without_Null_Unpadded);
  end SQL_VarChar_Ops;
private
  type SQL_VarChar is record
    Is_Null : Boolean := True;
    Text : SQL_VarChar_Not_Null;
  end record;
end SQL_VarChar;
```

C.2.9 SQL_Bit_Pkg

```
with SQL_System; use SQL_System;
with SQL_Boolean_Pkg; use SQL_Boolean_Pkg;
with Interfaces.SQL;
package SQL_Bit_Pkg is
  subtype SQL_Bit_Length is natural
    range 1 .. MAXCHRLen;
  subtype SQL_Unpadded_Length is natural
    range 0 .. MAXCHRLen;
  type SQL_Bit_Not_Null is new Interfaces.SQL.Bit;
  type SQL_Bit (Length : SQL_Bit_Length) is limited private;
  function Null_SQL_Bit return SQL_Bit;
  pragma INLINE (Null_SQL_Bit);
  -- the next three functions convert between
  -- null-bearing and not null-bearing-types
  -- Without_Null_Base and With_Null_Base are
  -- inverses (mod. null values)
  -- see also SQL_Bit_Ops generic package below
  function With_Null_Base (Value : SQL_Bit_Not_Null)
    return SQL_Bit;
  pragma INLINE (With_Null_Base);
  -- Without_Null_Base and Without_Null_Base_Unpadded raise
  -- null_value_error on the null input
```

```

function Without_Null_Base(Value : SQL_Bit)
    return SQL_Bit_Not_Null;
pragma INLINE (Without_Null_Base);
procedure Assign(
    Left : out SQL_Bit;
    Right : SQL_Bit);
pragma INLINE (Assign);
-- Substring(x,k,m) returns the substring of x starting
-- at position k (relative to 1) with length m.
-- returns null value if x is null
-- raises constraint_error if Start < 1 or Length < 1 or
-- Start + Length - 1 > x.Length
function Substring (
    Value : SQL_Bit;
    Start, Length : SQL_Bit_Length)
    return SQL_Bit;
pragma INLINE (Substring);
-- "&" returns null if either parameter is null;
-- otherwise performs concatenation in the usual way,
-- preserving all blanks.
-- may raise constraint_error implicitly if result is
-- too large (i.e., greater than SQL_Bit_Length'Last
function "&" (Left, Right : SQL_Bit)
    return SQL_Bit;
pragma INLINE ("&");
-- Logical Operations --
-- type X type => Boolean_with_unknown --
--the comparison operators return the boolean value
--UNKNOWN if either parameter is null; otherwise,
--the comparison is done in accordance with
--the rules of SQL 9075:1192 8.2
function Equals (Left, Right : SQL_Bit)
    return Boolean_with_Unknown;
pragma INLINE (Equals);
function Not_Equals (Left, Right : SQL_Bit)
    return Boolean_with_Unknown;
pragma INLINE (Not_Equals);
function "<" (Left, Right : SQL_Bit) return Boolean_with_Unknown;
pragma INLINE ("<");
function ">" (Left, Right : SQL_Bit) return Boolean_with_Unknown;
pragma INLINE (">");
function "<=" (Left, Right : SQL_Bit) return Boolean_with_Unknown;
pragma INLINE ("<=");
function ">=" (Left, Right : SQL_Bit) return Boolean_with_Unknown;
pragma INLINE (">=");
-- type => Boolean --
function Is_Null(Value : SQL_Bit) return Boolean;
pragma INLINE (Is_Null);
function Not_Null(Value : SQL_Bit) return Boolean;
pragma INLINE (Not_Null);
-- These functions of class type => Boolean
-- equate UNKNOWN with FALSE. That is, they return TRUE
-- only when the function returns TRUE. UNKNOWN and FALSE
-- are mapped to FALSE.
function "=" (Left, Right : SQL_Bit) return Boolean;
pragma INLINE ("=");
function "<" (Left, Right : SQL_Bit) return Boolean;
pragma INLINE ("<");
function ">" (Left, Right : SQL_Bit) return Boolean;
pragma INLINE (">");
function "<=" (Left, Right : SQL_Bit) return Boolean;
pragma INLINE ("<=");
function ">=" (Left, Right : SQL_Bit) return Boolean;
pragma INLINE (">=");
-- the purpose of the following generic is to generate
-- conversion functions between a type derived from
-- SQL_Bit_Not_Null, which are effectively Ada
-- arrays and a type derived from SQL_Bit, which

```

VERSION 2

```
-- mimic the behavior of SQL bit strings.
-- the subprogram formal is meant to default; that is,
-- this generic should be instantiated in the scope
-- of an use clause for SQL_Bit_Pkg.
generic
  type With_Null_Type is limited private;
  type Without_Null_Type is array (positive range <>)
    of Interfaces.SQL.Bitacter_Type;
  with function With_Null_Base (Value: SQL_Bit_Not_Null)
    return With_Null_Type is <>;
  with function Without_Null_Base (Value: With_Null_Type)
    return SQL_Bit_Not_Null is <>;
  with function Without_Null_Unpadded_Base (Value: With_Null_Type)
    return SQL_Bit_Not_Null is <>;
package SQL_Bit_Ops is
  function With_Null (Value : Without_Null_Type)
    return With_Null_Type;
  pragma INLINE (With_Null);
  function Without_Null (Value : With_Null_Type)
    return Without_Null_Type;
  pragma INLINE (Without_Null);
  function Without_Null_Unpadded (Value : With_Null_Type)
    return Without_Null_Type;
  pragma INLINE (Without_Null_Unpadded);
end SQL_Bit_Ops;
private
  type SQL_Bit (Length : SQL_Bit_Length) is record
    Is_Null: Boolean := True;
    Unpadded_Length: SQL_Unpadded_Length;
    Text: SQL_Bit_Not_Null(1 .. Length);
  end record;
end SQL_Bit_Pkg;
```

C.2.10 SQL_Enumeration_Pkg

```
with SQL_Boolean_Pkg; use SQL_Boolean_Pkg;
with SQL_Char_Pkg; use SQL_Char_Pkg;
generic
  type SQL_Enumeration_Not_Null is (<>);
package SQL_Enumeration_Pkg is
  ---- Possibly Null Enumeration ----
  type SQL_Enumeration is limited private;
  function Null_SQL_Enumeration return SQL_Enumeration;
  pragma INLINE (Null_SQL_Enumeration);
  -- this pair of functions convert between the
  -- null-bearing and not null-bearing types.
  function Without_Null (Value : in SQL_Enumeration)
    return SQL_Enumeration_Not_Null;
  pragma INLINE (Without_Null);
  -- With_Null raises Null_Value_Error if the input
  -- value is null
  function With_Null (Value : in SQL_Enumeration_Not_Null)
    return SQL_Enumeration;
  pragma INLINE (With_Null);
  procedure Assign (
    Left : in out SQL_Enumeration;
    Right : in SQL_Enumeration);
  pragma INLINE (Assign);
  -- Logical Operations --
  -- type X type => Boolean_with_unknown --
  -- these functions implement three valued logic
  -- if either input is the null value, the functions
  -- return the truth value UNKNOWN; otherwise they
  -- perform the indicated comparison.
  -- these functions raise no exceptions
  function Equals (Left, Right : SQL_Enumeration)
    return Boolean_with_Unknown;
```

```

function Not_Equals (Left, Right : SQL_Enumeration)
    return Boolean_with_Unknown;
pragma INLINE (Not_Equals);
function "<" (Left, Right : SQL_Enumeration)
    return Boolean_with_Unknown;
function ">" (Left, Right : SQL_Enumeration)
    return Boolean_with_Unknown;
function "<=" (Left, Right : SQL_Enumeration)
    return Boolean_with_Unknown;
function ">=" (Left, Right : SQL_Enumeration)
    return Boolean_with_Unknown;
-- type => Boolean --
function Is_Null (Value : SQL_Enumeration) return Boolean;
pragma INLINE (Is_Null);
function Not_Null (Value : SQL_Enumeration) return Boolean;
pragma INLINE (Not_Null);
function "=" (Left, Right : SQL_Enumeration) return Boolean;
pragma INLINE ("=");
function "<" (Left, Right : SQL_Enumeration) return Boolean;
pragma INLINE ("<");
function ">" (Left, Right : SQL_Enumeration) return Boolean;
pragma INLINE (">");
function "<=" (Left, Right : SQL_Enumeration) return Boolean;
pragma INLINE ("<=");
function ">=" (Left, Right : SQL_Enumeration) return Boolean;
pragma INLINE (">=");
-- the following six functions mimic the
-- 'Pred, 'Succ, 'Image, 'Pos, 'Val, and 'Value
-- attributes of the SQL_Enumeration_Not_Null type, passed
-- in, for the associated SQL_Enumeration (null) type
-- they all raise the Null_Value_Error exception if a null
-- value is passed in
-- Pred raises the Constraint_Error exception if the value
-- passed in is equal to SQL_Enumeration_Not_Null'Last
-- Succ raises the Constraint_Error exception if the value
-- passed in is equal to SQL_Enumeration_Not_Null'First
-- Val raises the Constraint_Error exception if the value passed
-- in is not in the range P'POS(P'FIRST)..P'POS(P'LAST) for
-- type P
-- Value raises the Constraint_Error exception if the sequence of
-- characters passed in does not have the syntax of an
-- enumeration literal for the instantiated enumeration type
function Pred (Value : in SQL_Enumeration)
    return SQL_Enumeration;
pragma INLINE (Pred);
function Succ (Value : in SQL_Enumeration)
    return SQL_Enumeration;
pragma INLINE (Succ);
function Pos (Value : in SQL_Enumeration) return Integer;
pragma INLINE (Pos);
function Image (Value : in SQL_Enumeration) return SQL_Char;
function Image (Value : in SQL_Enumeration_Not_Null)
    return SQL_Char_Not_Null;
pragma INLINE (Image);
function Val (Value : in Integer) return SQL_Enumeration;
pragma INLINE (Val);
function Value (Value : in SQL_Char) return SQL_Enumeration;
function Value (Value : in SQL_Char_Not_Null)
    return SQL_Enumeration_Not_Null;
pragma INLINE (Value);
private
type SQL_Enumeration is record
    Is_Null: Boolean := True;
    Value: SQL_Enumeration_Not_Null;
end record;
end SQL_Enumeration_Pkg;

```

Annex D Transform Chart

Function	sect.	input	output	output is
AdaID	5.3	Identifier	Ada Identifier	Delimited ident without quotes
AdaNAME	7.1.5	Record Declaration	Ada Identifier	Default name of the row record formal parameter
	8.6	Parameter	Ada Identifier	Name of the parameter in the Ada procedure declaration
	8.7	Select Parameter	Ada Identifier	Name of the component in the Ada row record type
	8.8	Insert Column Specification	Ada Identifier	Name of the component in the Ada row record type
	8.10	Value Expression	Ada Identifier	Default name for record component if expression appears as select parameter
AdaTYPE	7.1.4	Constant Declaration	Ada Identifier	Name of the type in Ada declaration of constant
	8.6	Parameter	Ada Identifier	Name of the type of the parameter in the Ada procedure declaration
	8.7	Select Parameter	Ada Identifier	Name of the type of the component in the Ada row record type
	8.8	Insert Column Specification	Ada Identifier	Name of the type of the component in the Ada row record type
COMP _{Ada}	8.2	Select Parameter	Ada Record Component	Used in declaration of the Ada row record type
		Insert Column Specification		
	8.4	Select Parameter	Ada Record Component	Used in declaration of the Ada row record type

VERSION 2

Function	sect.	input	output	output is
DATACLASS	5.4	Literal	Data Class	Data class of the literal
	7.1.1	Base Domain	Data Class	Data class of the base domain
	7.1.3	Domain	Data Class	Data class of the domain
		Domain Parameter		or domain parameter
	7.1.4	Constant	Data Class	Data class of the constant
	7.2	Column	Data Class	Data class of the column
	8.6	Parameter	Data Class	Data class of the parameter
	8.10	Value Expression	Data Class	Data class of the expression
DBLngNAME	8.7	Dblength Phrase	Ada Identifier	Name of dblength parameter
	7.1.5			undefined if no dblength phrase
DBLengAda	8.2	Select Parameter	Ada Record Component	Row record component
	8.4			used for dblength data
DBMS_TYPE	7.1.3	Domain	SQL Data Type	SQL data type to be
				used at the database interface with
				an object of the specified domain
DOMAIN	7.1.3	Domain Parameter	NO_DOMAIN	NO_DOMAIN is the domain
	7.1.4	Constant	Domain	of a domain parameter
				Domain of the constant
				(NO_DOMAIN for universal constants)
	7.2.1	Column	Domain	Domain of the column
	8.6	Parameter	Domain	Domain of the parameter
	8.10	Value Expression	Domain	Domain of the expression
				(NO_DOMAIN for literals)
INDICNAME	8.6	Parameter	SQL Identifier	Name of an SQL indicator parameter
	8.7	Select Parameter	SQL Identifier	Name of an SQL indicator parameter
	8.8	Insert Column Specification	SQL Identifier	Name of an SQL indicator parameter

Function	sect.	input	output	output is
INDIC _{SQL}	8.6	Parameter	SQL Indicator Parameter	An SQL indicator parameter for parameter in SQL statement
	8.7	Select Parameter	SQL Indicator Parameter	An SQL indicator parameter for parameter in SQL select list
	8.8	Insert Column Specification	SQL Indicator Parameter	An SQL indicator parameter for column in SQL statement
INTER_TYPE	7.1.3	Domain	SQL Data Type	Data type of SQL parameters
MODE	8.6	Parameter	Ada Mode	Mode of the parameter in the Ada procedure declaration
PARM _{Ada}	8.6	Parameter	Ada Parameter Declaration	Ada procedure declaration
PARM _{Row}	8.9	Into Clause	Ada Identifier	Ada parameter declaration in the Ada procedure declaration
		Insert From Clause		Name of row record parameter
PARM _{SQL}	8.6	Parameter	SQL Parameter	An SQL parameter declaration
	8.7	Select Parameter	SQL Parameter	An SQL parameter declaration
	8.8	Insert Column Specification	SQL Parameter	An SQL parameter declaration
QUALIFIER	5.4	Literal	SQL_interval_qualifier	Used for testing compatibility
	7.1.3	Domain	SQL_interval_qualifier	Used for testing compatibility
SQL_NAME	5.3	Identifier	SQL Identifier	Unique SQL identifier
	8.4	Cursor Name	SQL Identifier	Unique SQL cursor name
	8.6	Parameter Name	SQL Identifier	Unique SQL parameter name
	8.7	Select Parameter	SQL Identifier	Unique name of SQL parameter
	8.8	Column Name	SQL Identifier	Unique name of SQL parameter
SQL _{sc}	8.11	Search Condition	SQL Search Condition	An SQL search condition
SQL _{sq}	8.12	Subquery	SQL Subquery	An SQL subquery
SQL _{ve}	8.10	Value Expression	SQL Value Expression	An SQL value expression
TYPE _{Row}	8.9	Into Clause	Ada Identifier	Name of the type of the row record parameter
		Insert From Clause		Value assigned to the expression by the rules of SQL
VALUE	7.1.4	Static Expression	Ada Literal	

Annex E Glossary

Abstract interface. A set of Ada package specifications containing the type and procedure declarations to be used by an Ada application program to access the database.

Abstract module. A module that specifies the database routines needed by an Ada application.

Assignment context. A value expression appears in an assignment context if the value of that value expression is to be implicitly or explicitly assigned to an object. The assignment contexts are: select parameters, constant declarations, values in a VALUES list of an insert statement, set items in an update statement.

Base domain. A template for defining domains.

Conform. A value expression in an assignment context conforms to a target domain if the rules of SQL allow the assignment of a value of the data class of the expression to an object of the data class of the domain.

Conversion method. A method of converting non-null data between objects of the not null-bearing type, the null-bearing type, and the database type associated with the domain.

Correlation name. See ISO/IEC 9075:1992.

Cursor. See ISO/IEC 9075:1992.

Data class. The data class of a value is either character, integer, fixed, float, or enumeration. The data class of a domain determines which values may be converted, implicitly or explicitly, to the domain.

Database type. The SQL data type to be used with an object of that domain when it appears in an SQL parameter declaration. This need not be the same as the type of the data as is stored in the database.

Definitional module. A module that contains shared definitions: that is, declarations of base domains, domains, subdomains, constants, records, exceptions, enumerations, and status maps that are used by other modules.

Domain. The set of values and applicable operations for objects associated with a domain. A domain is similar to an Ada type.

Exposed. The exposed name of a module (table or view) that appears in a context clause (table ref) containing an as phrase (correlation name) is the identifier in the associated as phrase (correlation name); the module name (table name) is hidden. If the as phrase (correlation name) is not present, the exposed name is that module's (table's or view's) name. The exposed name of a table or module is the name by which that table or module is referenced.

Extended. A table, view, module, procedure, cursor, or cursor procedure that includes some nonstandard operation or feature.

VERSION 2

Hidden. See exposed.

Module. A definitional module, a schema module, or an abstract module.

Not null type. The Ada type associated with objects of a domain that may not take a null value.

Null type. The Ada type associated with objects of a domain that may take a null value.

Null value. SQL's means of recording missing information. A null value in a column indicates that nothing is known about the value that should occupy the column.

Options. The aspects of the base domain that are essential to the declaration of domains based upon the base domain. In particular they define the base domain's null- and not null-bearing type name, data class, database type, and conversion methods.

Patterns. A template used to create the Ada constructs that implement the Ada semantics of a domain, sub-domain, or derived domain declaration.

Row record. The Ada record associated with procedures that contain either a fetch, select, insert statement. It is used to transmit the database data to or from the client program.

Row record type. The Ada type of the row record.

SAME. SQL Ada Module Extensions.

SAMeDL. SQL Ada Module Description Language.

Schema module. A module containing definitions of database tables and views.

SQL. Formerly an acronym for Structured Query Language, SQL is the name of the relational database language defined in ISO/IEC 9075:1992.

SQLCODE. See ISO/IEC 9075:1992.

Standard Map. The Standard Map is a status map defined in SAMeDL_Standard that has the form "status Standard_Map as is_found uses boolean is (0 => true, 100 => false);". Standard_Map is the status map for fetch statements that appear in cursor declarations by default.

Standard post processing. The processing that occurs after the execution of an SQL procedure but before control is returned to the calling application.

Static expression. A value expression that can be evaluated at compile time (i.e., all the associated leaves consist solely of literals, constants, or domain parameter references).

VERSION 2

Status map. A partial function that associates an enumeration literal or a raise statement with each specified list or range of SQLCODE values. Status maps are used within the abstract module to uniformly process the status data for all procedures.

Target domain. The domain of the object to which an assignment is being made in an assignment context.

Universal constant. A constant whose declaration does not contain a domain reference.

Value expression. A value expression differs from an SQL value expression in that (1) an operand may be a reference to a constant or a domain parameter, and (2) SAMeDL value expressions are strongly typed.

Index

- Abstract Module 11, 12, 15, 16, 32, 36, 37, 38, 39, 45, 53
- abstract module 63
- Actual Identifier 10
- Ada Exception 18, 32
- Ada Identifier 2, 9
- AdaID 9, 10, 22, 28, 29, 31, 38, 39, 52, 54, 55, 57
- AdaNAME 29, 39, 40, 45, 46, 49, 52, 53, 54, 55, 57, 59, 60
- AdaTYPE 28, 39, 40, 45, 46, 49, 52, 53, 55
- As Phrase 2, 12
- Atomic Predicate 19
- Authorization Clause 15

- Bas Dom Ref 24
- Base Domain 12, 19, 20, 21, 22, 23, 24, 26, 36, 37
- Base Domain Body Declaration 19
- Base Domain Declaration 19, 20, 25
- Base Domain Name 20
- Base Domain Parameter 20, 24, 25
- Base Domain Reference 13, 24, 26
- Bit 10, 20, 21, 28
- Bit Literal 10

- Character 10, 20, 21, 28
- Character Literal 10, 21, 22, 31
- Close Statement 48, 50
- Column Definition 19, 34
- Column Name 13, 55
- Column Reference 14, 61
- Comment 11
- COMPAda 37, 40, 41, 46, 51
- Compilation Unit 11
- Component 29
- Component Declaration 29
- Component Declarations 29
- Component Name 29, 53
- Conform 28, 43, 44, 55
- Constant Declaration 19, 27, 28
- Constant Reference 13, 28, 41, 56, 61
- Context 12, 19, 33, 37
- Context Clause 11, 12, 13, 15
- Conversion Method 23, 36
- Converter 22
- Correlation Name 53
- Cursor 12, 15, 38, 42, 44, 45, 46, 49, 50
- Cursor Declaration 13, 33, 38, 44, 45, 46, 49, 50, 51, 54
- Cursor Delete Statement 48, 50
- Cursor Name 14, 18
- cursor name 48
- Cursor Proc Reference 13
- Cursor Procedure 14, 15, 38, 48, 49
- Cursor Procedures 44, 45, 46, 48, 49, 50
- Cursor Reference 13
- Cursor Statement 48, 49
- Cursor Update Statement 48, 49, 50

- Data Class 10, 20, 21, 22, 23, 24, 25
- Database Literal 10, 24, 25, 28, 61
- Database Mapping 21, 23, 24, 25
- DATACLASS 10, 25, 28, 34, 59, 60
- Date 10, 20
- Date Literal 10
- Datetime 10
- DBLengAda 37, 40, 41, 46, 51
- Dblength 29, 30, 40, 45, 53, 54
- DBLngNAME 45, 53, 54
- Dbms 22
- Dbms Type 22, 23
- DBMS_TYPE 24, 35
- Default Mapping 21
- Defining Location 13
- Definition 19, 20
- Definitional Module 11, 12, 15, 19, 20, 26
- Delete Statement 13
- Delimited Identifier 10
- Derived Domain Pattern 21
- Dom Ref 24
- DOMAIN 15, 28, 34, 44, 45, 53, 55, 56, 59
- Domain Conversion 61
- Domain Declaration 19, 21, 22, 24, 25, 26
- Domain Parameter Reference 13, 28, 61
- Domain Pattern 21
- Domain Reference 2, 13, 24, 27, 28, 29, 51, 52, 59

- Enumeration 10, 20, 21, 24, 25, 28, 30, 62
 - Association 24, 25, 26
 - Association List 24, 25
 - Declaration 19, 30
 - Literal 10, 13, 18, 21, 25, 28, 30, 31, 32, 36, 37, 56, 61
 - Literal List 30, 31
 - Literal Reference 13
 - Reference 13, 21, 24, 25, 32
- Enumeration Declaration 30
- Enumeration literal 30
- Enumeration literal list 30
- Exception 13, 18, 19, 31
- Exception Declaration 19, 31
- Exception Reference 13
- Exposed 12, 15

VERSION 2

Extended 18, 23, 33, 34, 37, 38, 39, 44, 45, 48, 49, 52, 64

Extended Cursor Statement 18, 48, 49

Extended Query Expression 18, 44, 45

Extended Query Specification 18

Extended Schema Element 18, 34

Extended Statement 18, 39, 40, 41, 51

Extended Table Element 18, 19

Fetch Statement 33, 48, 49, 50, 51, 53, 54, 57

Fixed 10, 20

Fixed Literal 10

Float 10, 20

Float Literal 10

From Clause 13, 53, 57

Function 22

Fundamental 22

Hidden 12, 16

Identifier 2, 10, 13, 19, 20, 24, 27, 29, 30, 31, 32, 33, 37, 38, 39, 44, 46, 48, 51, 57

Image 20, 21, 24, 26

Indicator Parameter 36, 46, 51, 52, 54, 56, 61

indicator parameter 37

INDICNAME 46, 52, 54, 56, 61

INDICSQL 41, 46, 51, 52, 54, 56, 61

Input Parameter 2, 13, 15, 16

Input Parameter Declaration 2

Input Parameter List 19, 39, 41, 44, 46, 48, 49, 50, 51

Input Reference 13

Insert Column List 40, 41, 43, 44, 55, 56

Insert Column Specification 55

Insert From Clause 40, 43, 57

Insert Statement 13, 43

Insert Statement Row 40, 41, 43, 44, 55, 57

Insert Value 55

Insert Value List 40, 41, 43, 44, 55, 56

Integer 10, 20

Integer Literal 10

INTER_TYPE 24, 46, 52, 54

Interface Type 22, 23

Interval 10, 20

Interval Qualifier 24

Into Clause 19, 39, 43, 44, 49, 57

Item 3, 12, 13, 15, 16

Left Hand Side 17, 32

LENGTH 21, 24, 25

Enumeration 28

Literal 10, 23, 27, 40, 41, 55, 56, 60

Map 20, 21, 23, 24, 25

Mode 19, 39, 40, 49, 50

Module 11, 12, 13, 16, 18, 28, 29, 31, 36, 45

Module Name 12, 13, 14, 15, 18

Module Reference 12, 13

Named Phrase 29, 32, 51, 53, 55, 56, 62

NO_DOMAIN 25, 53

Not Null 2, 22, 23, 24, 26, 28, 29, 30, 36, 37, 45, 51, 52, 53, 54, 55, 56, 61

Not Null Only 26, 30, 44, 45, 52, 53, 55, 56

Null 4, 22, 23, 30, 36, 37, 44, 52, 53, 55, 56

Null Value Error Exception 37, 65

Numeric Literal 10

Open Statement 48, 49, 50

Options 22, 23, 24

Parameter 15, 22, 23, 51, 52

Parameter Association 23, 24, 25

Parameter Association List 20, 24

Parameter Declaration 20

Parameter Name 20

Parent 24, 25

PARMAda 39, 41, 49, 51

PARMRow 40, 49, 57

PARMSQL 41, 46, 51, 52, 54, 56, 61

Pattern 21, 22, 23, 26

Pattern Element 21

Pattern List 21

Patterns 21

Poor Programming Practice 43, 53

Pos 21, 24, 25

Procedure 22

Procedure Declaration 38, 39, 40, 41, 45, 49, 50, 53

Procedure Name 18

Procedure Or Cursor 13, 38

Procedure Reference 13

Qualifier 21, 25

Query 44, 45

Query Specification 53

Raise 18, 32

Record Declaration 19, 29

Record Id 57

Record Reference 13, 57

Reference 14, 16

Reference Location 13, 14, 15

Row Record 39, 40, 41, 49, 51, 55, 57, 59

VERSION 2

SCALE 21, 24, 25, 59
Schema 11, 12, 13, 15, 16
Schema Element 33
Schema Module 11, 12, 33, 34
Schema Name 12, 13, 53
Schema Reference 13
Scope 12, 15, 16
Select List 40, 41, 44, 46, 50, 53, 54
Select Parameter 53
select parameter 37
Select Statement 39, 40, 41, 43, 44, 54, 57
Set Item 44, 49, 50
SQL Bit String Literal 10
SQL Close Statement 50
SQL Data Type 56
SQL Database Error Exception 18, 65
SQL Delete Statement 50
SQL Fetch Statement 50
SQL Fetch Target List 54
SQL hex string literal 10
SQL Identifier 2, 13, 56
SQL Insert Column List 56
SQL Insert Statement 44
SQL Insert Value List 56
SQL interval literal 10
SQL Select List 44, 46, 54
SQL Select Statement
 Single Row 44
SQL Select Target List 54
SQL Status Parameter 17, 18, 32, 41, 51
SQL Target Specification 46, 54
SQL Update Statement Positioned 50
SQL Value Expression 61
SQL_Standard 29
SQLCODE 18, 32, 35, 41, 50
SQLNAME 10
SQLNAME 46, 50, 52, 54, 56, 61
SQLSC 35, 41, 44, 47, 62
SQLSTATE 17, 32, 35, 41, 50
SQLVE 35, 37, 41, 44, 50, 56, 59, 61, 62
Standard Map 33, 48
Standard Post Processing 17, 18, 32, 41, 51
Standard_Map 33
Statement 36, 39, 40
Static Expression 20, 24, 25, 27, 28, 32, 37
Status Assignment 17, 18, 32, 33
Status Clause 17, 18, 34, 35, 39, 40, 41, 48, 50, 62, 63
Status Map 12, 17, 18, 19, 32, 33, 62, 63
Status map 30
Status Map Declaration 19, 32
Status Parameter 18, 40, 50

Status Reference 13, 62
Subdomain Declaration 19, 24, 25, 26
Subdomain Pattern 21
Subdomain Reference 13, 24
Subquery 43, 45

Table Element 19
Table Name 13, 18, 43, 53
Table Reference 13, 14
Target Enumeration 32
Target enumeration 32
Temporal 28
Time 10, 20
Time Literal 10
Timestamp 10, 20
Timestamp Literal 10
Type 22
Type Mark 22
TYPERow 40, 49, 57

Universal 28
Update Statement 13, 39, 44, 49
Use Clause 12, 16

VALUE 28
Value Expression 27, 44, 53, 59, 61
View Name 18
Visible 16

With Clause 12, 15
With Schema Clause 12, 15, 19
Word List 22

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION Unclassified			1b. RESTRICTIVE MARKINGS None		
2a. SECURITY CLASSIFICATION AUTHORITY N/A			3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for Public Release Distribution Unlimited		
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE N/A					
4. PERFORMING ORGANIZATION REPORT NUMBER(S) CMU/SEI-95-SR-018			5. MONITORING ORGANIZATION REPORT NUMBER(S)		
6a. NAME OF PERFORMING ORGANIZATION Software Engineering Institute		6b. OFFICE SYMBOL (if applicable) SEI	7a. NAME OF MONITORING ORGANIZATION SEI Joint Program Office		
6c. ADDRESS (city, state, and zip code) Carnegie Mellon University Pittsburgh PA 15213			7b. ADDRESS (city, state, and zip code) HQ ESC/ENS 5 Eglin Street Hanscom AFB, MA 01731-2116		
8a. NAME OFFUNDING/SPONSORING ORGANIZATION SEI Joint Program Office		8b. OFFICE SYMBOL (if applicable) ESC/ENS	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER F19628-95-C-0003		
8c. ADDRESS (city, state, and zip code)) Carnegie Mellon University Pittsburgh PA 15213			10. SOURCE OF FUNDING NOS.		
			PROGRAM ELEMENT NO 63756E	PROJECT NO. N/A	TASK NO N/A
11. TITLE (Include Security Classification) Information Technology -- Programming Language -- The SQL Ada Module Description Language (SAMeDL)					
12. PERSONAL AUTHOR(S) Marc Graham					
13a. TYPE OF REPORT Final		13b. TIME COVERED FROM TO		14. DATE OF REPORT (year, month, day) October	
15. PAGE COUNT 110					
16. SUPPLEMENTARY NOTATION					
17. COSATI CODES			18. SUBJECT TERMS (continue on reverse of necessary and identify by block number) Ada, ISO, SQL		
FIELD	GROUP	SUB. GR.			
19. ABSTRACT (continue on reverse if necessary and identify by block number)					
(please turn over)					
20. DISTRIBUTION/AVAILABILITY OF ABSTRACT UNCLASSIFIED/UNLIMITED <input checked="" type="checkbox"/> SAME AS RPT <input type="checkbox"/> DTIC USERS <input checked="" type="checkbox"/>			21. ABSTRACT SECURITY CLASSIFICATION Unclassified, Unlimited Distribution		
22a. NAME OF RESPONSIBLE INDIVIDUAL Thomas R. Miller, Lt Col, USAF			22b. TELEPHONE NUMBER (include area code) (412) 268-7631		22c. OFFICE SYMBOL ESC/ENS (SEI)